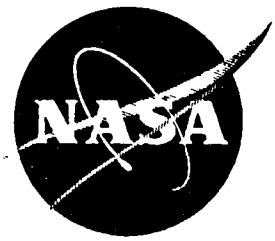
NASA-CR-168019
19830005868

NASA CR-168019
MTI 83TR12



DIGITAL SYSTEM FOR STRUCTURAL DYNAMICS SIMULATION

by A.I. Krauter, L.J. Lagace, M.K. Wojnar and C. Glor

SHAKER RESEARCH CORPORATION

prepared for

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

NASA Lewis Research Center
Contract NAS 3-22546

1 Report No	2 Government Accession No	3 Recipient's Catalog No
NASA CR 168019		

4 Title and Subtitle	5 Report Date
Digital System for Structural Dynamics Simulation	November, 1982
	6 Performing Organization Code

7 Author(s)	8 Performing Organization Report No
A. I. Krauter, L. J. Lagace, M. K. Wojnar, C. Glor	83TR12
	10 Work Unit No

9 Performing Organization Name and Address	11 Contract or Grant No
Shaker Research Corp. 968 Albany-Shaker Road Latham, New York 12110	NAS3-22546
	13 Type of Report and Period Covered
12 Sponsoring Agency Name and Address	Contractor Report
NASA Lewis Research Center 21000 Brookpark Rd Cleveland, OH 44135	14 Sponsoring Agency Code

15 Supplementary Notes

Final Report Project Manager: L. J. Kiraly
MS 23-2
NASA Lewis Research Center

16 Abstract

The objective of this program, was to develop and document the design of a digital device for structural dynamics simulation. The simulator design has incorporated state-of-the-art digital hardware and software for the simulation of complex structural dynamic interactions, such as those which occur in rotating structures. The targeted uses of this system include simulations and parametric design studies to identify improved design criteria and methodology, to identify structural dynamics instabilities and to evaluate the effects of local non-linearities, transient loadings, and engine control instabilities.

The system has been designed to use an array of processors wherein the computation for each physical subelement or functional subsystem would be assigned to a single specific processor in the simulator. These node processors are custom designed microprogrammed bit-slice microcomputers which function autonomously and can communicate with each other and a central control minicomputer over parallel digital lines. Inter-processor nearest neighbor communications busses pass the constants which represent physical constraints and boundary conditions. Each node processor has its own program and data memory. Each node processor calculates its results independently and simultaneously with the other node processors. The node processors are connected to the six nearest neighbor node processors to simulate the actual physical interface of real substructures.

Computer generated finite element mesh and force models can be developed with the aid of the central control minicomputer. The program so developed is converted to the proper format, segmented and loaded into the individual processors which make up the simulator. The control computer also oversees the animation of a graphics display system, disk-based mass storage along with the individual processing elements.

The mathematical approach to simulating the dynamic behavior of an engine system is based upon the explicit time integration of the state vectors assigned to the individual processors. An integration technique such as fourth order Runge-Kutta is applicable to this analysis.

17 Key Words (Suggested by Author(s))	18 Distribution Statement
Simulation, High Speed Rotating System Simulation, Rotating Structures Simulation Parallel, Processing Methods, Digital Processor Arrays	Unclassified, Unlimited

19 Security Classif (of this report)	20 Security Classif (of this page)	21 No of Pages	22 Price*
Unclassified	Unclassified	125	

* For sale by the National Technical Information Service, Springfield, Virginia 22161

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

SUMMARY

The objective of this program, conducted by Shaker Research Corporation, was to develop and document the design of a digital device for structural dynamics simulation. The simulator design has incorporated state-of-the-art digital hardware and software for the simulation of complex structural dynamic interactions, such as those which occur in rotating structures. The targeted uses of this system include simulations and parametric design studies to identify improved design criteria and methodology, to identify structural dynamics instabilities and to evaluate the effects of local non-linearities, transient loadings, and engine control instabilities.

The system has been designed to use an array of processors wherein the computation for each physical subelement or functional subsystem would be assigned to a single specific processor in the simulator. These node processors are custom designed microprogrammed bit-slice microcomputers which function autonomously and can communicate with each other and a central control minicomputer over parallel digital lines. Inter-processor nearest neighbor communications busses pass the constants which represent physical constraints and boundary conditions. Each node processor has its own program and data memory. Each node processor calculates its results independently and simultaneously with the other node processors. The node processors are connected to the six nearest neighbor node processors to simulate the actual physical interface of real substructures.

Computer generated finite element mesh and force models can be developed with the aid of the central control minicomputer. The program so developed is converted to the proper format, segmented and loaded into the individual processors which make up the simulator. The control computer also oversees the animation of a graphics display system, disk-based mass storage along with the individual processing elements.

The mathematical approach to simulating the dynamic behavior of an engine system is based upon the explicit time integration of the state vectors assigned to the individual processors. An integration technique such as fourth order Runge-Kutta is applicable to this analysis.

The hardware was designed as an array of 125 processors in a cubic structure. Each node processor was designed for very high speed multiplication and addition which are fundamental requirements for the time-step integration algorithms. This implementation of an array of processors operating in parallel has the capability of solving simulation problems an order of magnitude faster than a conventional serial computer.

INTRODUCTION

This program was initiated to investigate and document the design of a Digital System for Structural Dynamics Simulation. The intended use for the system is a design and analysis aid for the production of gas turbine engines. This simulator would realize both a savings of money and of time by reducing the need for extensive prototyping during the development of gas turbine engine hardware. While simulation methods exist for use on main frame computers, this system uses the architecture of an array of processors for greater throughput.

Current and on-going work in the field of parallel processing and simulation was reviewed and is summarized in Appendices I and II.

Special acknowledgement is given to Mr. L. James Kiraly, Project Manager of the Digital System for Structural Dynamics Simulation at NASA-Lewis Research Center. Mr. Kiraly's contributions were pertinent to the success of this effort. His ideas are reflected throughout this final report.

The scope of this work included the study of simulation techniques useful with parallel processors, the design of the system architecture, the hardware design of the individual processors in the array (node processors), and the design and flowcharting of the processor instructions. Special emphasis was placed upon the design of the system architecture and the node processor hardware. Particular attention was paid to the high risk areas of the node processor design. For instance, floating point hardware for multiplication, addition, subtraction and division was designed through detailed schematic diagrams. The CPU and communications controller, being more conventional, were detailed only to the block diagram stage. The node processor memory was designed to the schematic level.

Software design centered on the node processor. A very large and powerful custom instruction set tied intimately to simulation techniques was developed. The concepts necessary for on line and off line software were started, but no attempt was made to write this software.

The concepts of simulation with regard to an array of processor solution were developed in broad terms. Segmentation of a problem, problem size and computational requirements were carried to the level sufficient for node processor definition.

For a successful implementation of this system, follow-on is needed in all three of the above areas. A sample problem must be developed and programmed on a main frame computer. Further hardware detailing is necessary to complete the node processor design. A breadboard version of a node processor should be constructed, microcoded, and programmed with the sample problem. Software development is needed to reach this prototype stage. Additional software development is then required to implement the entire array of processors.

The following section describes the simulation analysis leading to the system architecture and node processor architecture.

ANALYSIS - REQUIREMENTS FOR SIMULATION

In this section the groundwork for the Digital System for Structural Dynamics Simulation is developed. Two sample problems are discussed. The first problem describes a simplified nonlinear simulation problem and its solution on an array of processors. The second problem is linear and is used to set the bounds on the hardware by limiting the overall size of the physical model. Some additional detail on problem substructuring is brought out. A time step integration method appropriate for the solution is developed. Finally, the hardware and software requirements are outlined prior to their detailed discussion in the following two sections.

SAMPLE PROBLEM 1

This sample problem, based on work by J. J. Tomko, is used to illustrate techniques that can be used to solve a simulation problem on an array of processors with time step integration methods. Other models and techniques other than the ones presented for this model can also be employed. The model is shown in Figure 1. The model contains m disks including shaft segments which have mass. Each disk may contain n blades and there are b bearings.

Disk Representation

Each disk is represented by five coordinates. These coordinates are two translations and three rotations. Each disk is taken as rigid. The disk equation is of the form:

$$[M_{di}]\{\ddot{q}_{di}\} = f(t,\{\dot{q}_{di}\}, \{q_{di}\}, \{\dot{q}_{d\ i-1}\}, \{q_{d\ i-1}\}, \{\dot{q}_{d\ i+1}\}, \{q_{d\ i+1}\}$$
$$\{\dot{q}_{bij}\}, \{q_{bij}\}, \{\dot{q}_{ci}\}, \{q_{ci}\}) \qquad (1)$$

where $\{q_{di}\}$ is the five element state vector for disk i

$\{q_{d\ i-1}\}$ is the five element state vector for disk i-1

$\{q_{d\ i+1}\}$ is the five element state vector for disk i+1

$\{q_{bij}\}$ is the state vector for blade j on disk i. This vector can contain 4 elements.

SAMPLE PROBLEM I

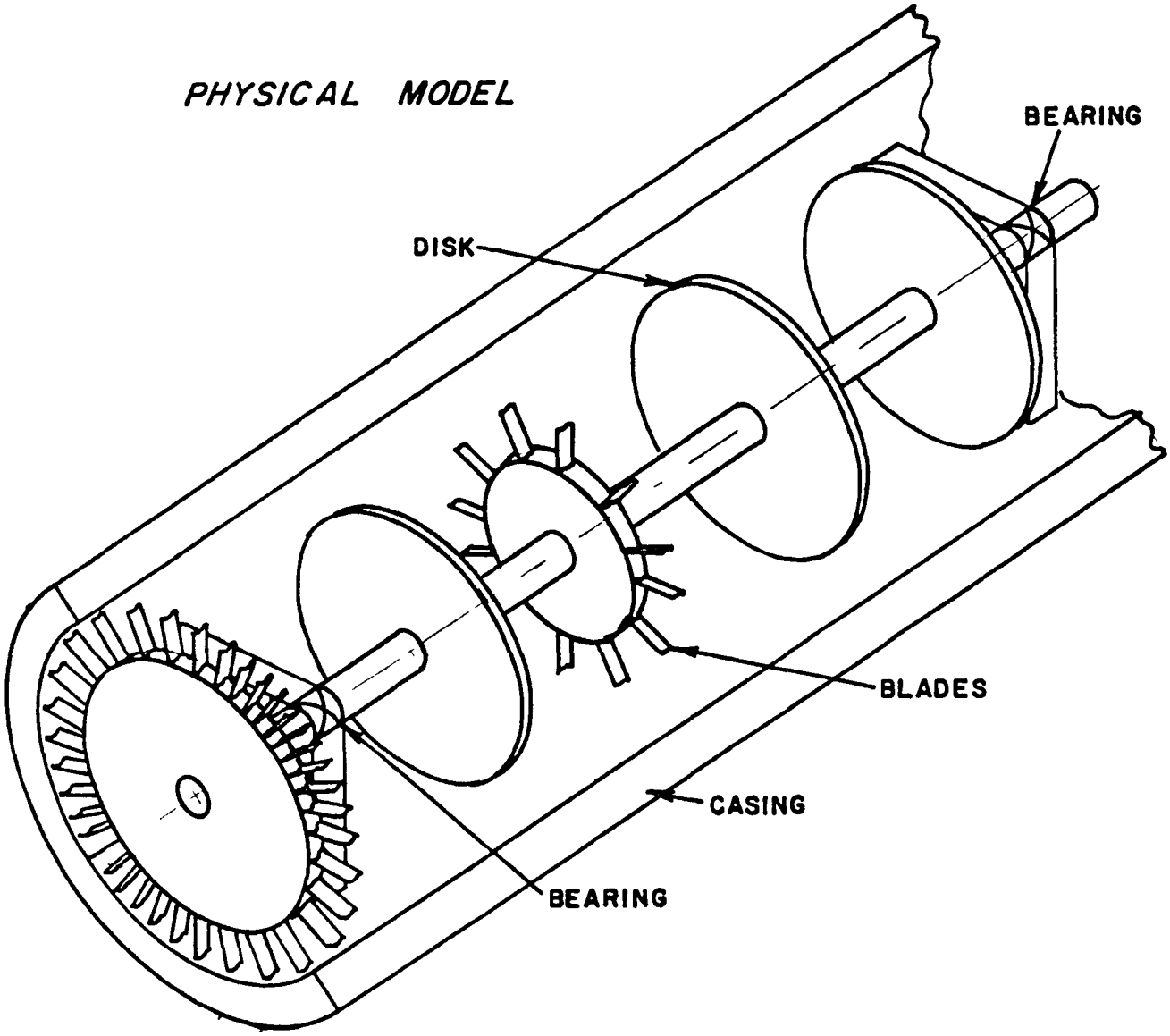


Figure 1 Model Containing m Disks (Including Shaft Segments Having Mass). Each Disk Can Contain n Blades. There are b Bearings. Model Based on That of Tomko, J.J.

$\{q_{ci}\}$ is the state vector for the casing at disk i. This vector can contain 4 elements at a bearing or 80 elements at a bladed disk.

$[M_{di}]$ can be non-diagonal and time variable. The non-diagonal character arises from the use of a point other than the mass center for the definition of the translational degrees of freedom. The time variation arises from this and from the use of fixed Cartesian coordinates.

Blade Representation

Each blade is represented by two lumped masses having two degrees of freedom each. These degrees of freedom are axial and circumferential translation.

The equation for each blade is of the form:

$$[M_{bij}]\ \{\ddot{q}_{bij}\} = F(t,\ \{\dot{q}_{bij}\},\ \{q_{bij}\},\ \{\dot{q}_{di}\},\ \{q_{di}\},\ \{\dot{q}_{ci}\},\ \{q_{ci}\}) \qquad (2)$$

where

$\{q_{bij}\}$ is the 4 element blade state vector, and where

$\{q_{ci}\}$ is the state vector for the casing at disk i. This vector can contain, say, 40 radial and 40 axial coordinates for a casing cross-section at a bladed disk.

The matrix $[M_{bij}]$ is assumed to be diagonal and constant — the equations for the blades must be written using inertial coordinates.

Casing Representation

Assume the casing has been modeled by a large scale finite element program which includes, say, 40 nodes at each disk. Each of these nodes can have radial and axial displacement. At bearing stations, the finite element program provides at least two displacements and two rotations.

Assume also, that the finite element program produces up to 10 modes. For

each mode, the mode vector is $\{u_r\}$, where this vector contains all of the points used in the finite element analysis. The vector $\{u_{ri}\}$ is that part of the modal motion occurring at cross-section i.

At disk i (i.e., cross-section i), the casing coordinates are given by

$$\{q_{ci}\} = \sum_{r=1}^{10} \{u_{ri}\}\, c_r(t) \quad \text{where } c_r(t) \text{ is the modal motion of mode r.}$$

The equation of motion for mode r is

$$\ddot{c}_r + 2\eta\omega_r\dot{c}_r + \omega_r^2 c_r = \sum_{i=1}^{m} \frac{\{u_{ri}\}^t\{f(t)_i\}}{\{u_r\}^t\,[M_c]\,\{u_r\}} \tag{3}$$

where η is a user-supplied modal damping factor, ω_i is the modal frequency, $\{f(t)\}$ is the force of the blades or of the bearing on the casing at station i, and $[M_c]$ is the mass matrix for the finite element model. Note that $\{u_r\}^T \cdot [M_c] \cdot \{u_r\}$ is a single scalar number for each mode and need be computed only once.

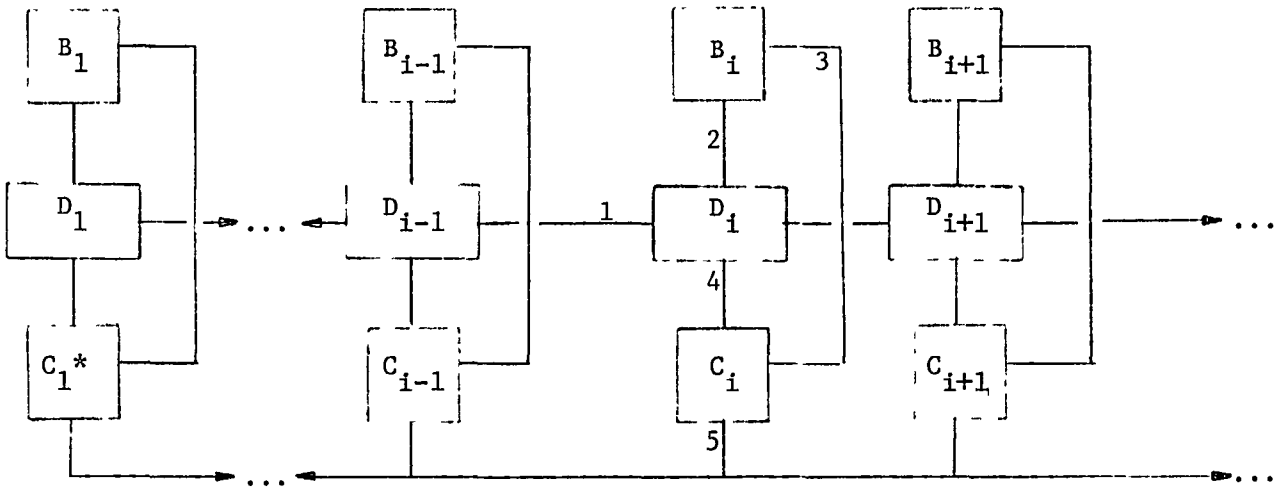
At each disk cross-section $\{u_{ri}\}$ and $\{f(t)_i\}$ can contain 80 elements. Consequently the $\{q_{ci}\}$ can also contain 80 elements.

The force $\{f(t)_i\}$ for blade j is, in general, represented by

$$\{f(t)_i\} = f_i(\{\dot{q}_{ci}\},\ \{q_{ci}\},\ \{\dot{q}_{bij}\},\ \{q_{bij}\})$$

A Parallel Processing Approach

One approach to simulate the motion of the engine (i.e., to solve the dynamic equations of motion) is as follows (in this, each box denotes a processor and each line denotes transmission of information between processors). The controlling minicomputer is not shown.



Typical Node Processor Assignment

The array of processor nodes are assigned such that

B_i denotes a processor that treats all the blades on a disk,

D_i denotes a processor that treats a disk or a lumped mass segment of the shaft, and

C_i denotes a processor that treats a portion of the casing (i.e., that portion of the casing associated with a disk or with a lumped mass shaft segment).

*The first casing processor has tasks beyond those of casing processors C_2 through C_m.

The functions of each processor are as follows:

PROCESSORS D_1 through D_{I+1}

In general, $[M_{di}]$ must be evaluated at each time step and then inverted

From adjacent D's, get $\{\dot{q}_{d\ i-1}\}$, $\{q_{d\ i-1}\}$, $\{\dot{q}_{d\ i+1}\}$, and $\{d_{d\ i+1}\}$.

From the B_i, get average of $\{\dot{q}_{bij}\}$, $\{q_{bij}\}$,

where $j = 1, \ldots, N_i$. (For bladed disk only.) Five averages are received.

From the casing processor, get $\{\dot{q}_{ci}\}$ and $\{q_{ci}\}$, (For bearings, all quantities may be needed to solve equation (1). For a bladed disk, only the radial displacements will be necessary. The forces of the blades on the disk in the radial direction can be obtained by curve fitting the best ellipse to the radial casing displacements and then evaluating blade interference via a tip circle approach.)

Equation (1) may then be solved.

PROCESSORS B_1 through B_{I+1}

For each blade $[M_{bij}]$ can be inverted once and stored if, as assumed, this matrix is constant (time-independent).

From disk, get $\{\dot{q}_{di}\}$ and $\{q_{di}\}$. (These are needed for the blade force calculation. The radial velocity of the disk is necessary to compute the Coriolis component of circumferential blade acceleration.)

From casing, get velocities and positions of curve-fitted averages of casing points (i.e., best ellipse in plane of cross-section and in an axial plane, etc.). Processor must then determine which blades are rubbing and what the circumferential and axial forces on those blades are.

Equation (2) may then be solved.

PROCESSORS C_1 through C_m

These processors do not solve modal state equation (3). (Only processor C_1 solves this state equation, see below.)

From blades (if present at disk i), get best tip ellipse in plane of cross-section and in an axial plane. (This tip circle is used for part of the computation of the forces of the blades on the casing.)

From disk get $\{\dot{q}_{di}\}$ and $\{q_{di}\}$. (If disk with blades is located at cross-section i, the radial displacement of the disk is used for the remaining part of the computation of the forces of the blades on the casing.)

From C_1, get $C_r(t)$, r = 1, . . ., $\overline{M}$ where $\overline{M}$ is the total number of casing modes (say 10). The processor C_1 contains mode shape information for this (the ith) cross-section. Therefore $\{\dot{q}_{ci}\}$ and $\{q_{ci}\}$ can be determined from $C_r(t)$.

The processor C_i determines force of either shaft on casing (using $\{\dot{q}_{di}\}$ and $\{q_{di}\}$) or of blades on casing (using these and $\{\dot{q}_{bij}\}$ and $\{q_{bij}\}$). For the latter, blade forces on casing are computed via the tip circle approach and via comparison of the best tip ellipses with the casing displacements. The forces are assigned to nearest casing nodes (nodes in the cross-section and circumferentially nearest to rubbing points). The processor C_i then determines contribution of this cross-section to the right side of equation (3).

Processor C_1 solves equation (3). Contributions to right side of (3) are received from processors C_i, $i = 2, \ldots, m$. The total for each mode is assembled and then each modal equation (3) is solved.

Number of Path Variables

- Time not included. Time is tracked by each processor. Timing is controlled by controlling minicomputer (not shown).
- Volume of data transmitted along a path is generally equal in both directions.

PATH 1 Assume rotor modeled as a beam with torsion. Have two displacements and two slopes plus torsion. Result is $5 \times 2^* = 10$ scalars.

PATH 2 Into D_i — 5 average displacements. Result is $5 \times 2^* = 10$ scalars.

PATH 3 Into B_i — best ellipses (4 coordinates for each plane). Result is $8 \times 2^* = 16$ scalars.

Into C_i — best ellipses (4 coordinates for each plane). Result is $8 \times 2^* = 16$ scalars.

PATH 4 Into C_i — say 4 degrees of freedom for bearing/casing interaction. Result is $4 \times 2^* = 8$ scalars.

PATH 5 Into C_1 — say 10 modes. Result is $10 \times 2^* = 20$ scalars.

From C_i — one contribution for each mode. Result is $10 \times 2^* = 20$ scalars.

*Factor 2 is for velocities.

Simulation Procedures

There are several choices in the method chosen for simulation. These choices fall into two categories, explicit methods and implicit methods.

Explicit methods require equations of motion in first order form. Velocities and accelerations are obtained at each time step and then integrated. The integration is separate from the equations of motion. These methods are relatively simple to implement. Two explicit methods are the Runge-Kutta and the Predictor-Corrector method.

In the Runge-Kutta method, more solutions of equations of motion are necessary at each time step for a given accuracy. There is no inherent measurement of error. No history of the problem is required. It is self starting, relatively simple and easy to change time steps.

In the Predictor-Corrector method, fewer solutions of equations of motion at each time step for a given measure of accuracy are necessary. There is an inherent measure of error. A history of the problem is required. It is not self starting, relatively complicated and relatively hard to change time steps.

Implicit methods require equations of motion in second order form. Velocities and accelerations are represented by finite differences so that displacements are obtained directly at each time step. The integration procedure is closely coupled with the equations of motion. These methods are relatively complicated.

The Runge-Kutta method has been chosen as the most suitable for the structural dynamics simulator because it is self starting, does not require a history and is relatively simple to program.

SAMPLE PROBLEM 2 - DETERMINATION OF PROCESSOR COMPUTATIONAL CAPABILITIES

This sample problem is linear and is developed to determine the functions, the memory, communications, instructions and speed desired of the parallel processing system.

Largest Linear problem

The physical problem discussed will be the largest linear problem that the system is designed to accept. Computational requirements for the linear problem can be assessed in a straightforward manner. Extent and nature of nonlinearities are not known so that their computational requirements cannot be established. However, computation capabilites for typical nonlinear calculations will be available since the actual associated linear problem will normally be much smaller than this largest linear problem. To allow for nonlinear computations, the following functions at a minimum, should be available:

$$\sin x,\ \tan^{-1}x, \sqrt{x}.$$

Less important, but also desirable, are the functions α^x and $\log_p x$ where α and x are real numbers and where p is either 10 or e.

The overall equation for the entire linear physical problem being simulated can be written in the form

$$[M]\ \{\ddot{q}\} + [C]\ \{\dot{q}\} + [K]\ \{q\} = \{f(t)\} \qquad (4)$$

where $\{q\}$ is a vector containing N elements. The matrices [M], [C], and [K] are the mass, damping, and stiffness matrices, respectively. The vector $\{f(t)\}$ contains N elements whose values, at any time, can be computed directly.

Time Dependence of Matrices

The matrices [M], [C], and [K] can contain elements whose values are functions of time.

This is true because elements of [M] which vary with time can arise for a certain choice in coordinate systems and in their associated coordinates {q}. Time variation in elements of [C] and [K] arise when problems containing parametric excitation are considered.

Banding of Matrices

The matrices [C] and [K] can be formulated such that they are banded.

These matrices will be banded for structural stiffness matrices since a generalized displacement applied at a node will produce forces at only that node and at its neighbors. (Proper node sequencing is required.)

Diagonal Nature of Mass Matrix

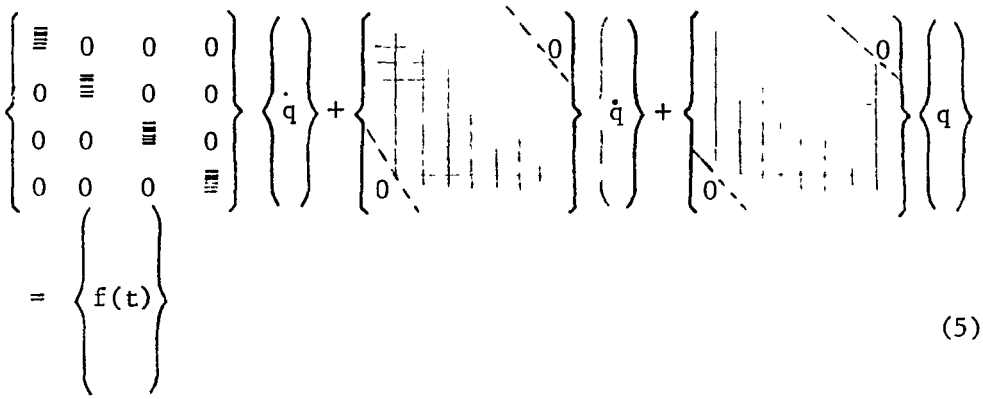
The matrix [M] is locally diagonal; i.e., the matrix has the form

$$\begin{Bmatrix} \blacksquare & 0 & 0 & 0 \\ 0 & \blacksquare & 0 & 0 \\ 0 & 0 & \blacksquare & 0 \\ 0 & 0 & 0 & \blacksquare \end{Bmatrix}$$

where the shaded regions only have non-zero elements.

This assumption is critical in that it provides for mass decoupling of regions of the problem. To produce such a mass matrix, each of these regions must be connected to the other regions only by members having stiffness and damping properties but no mass. In addition, the generalized coordinates used for each region of the problem must not reference coordinates in other regions of the problem.

With the above properties, Equation (4) can be put in the form



(5)

where the bandwidth of the second and third matrices is greater than the rank of the largest locally diagonal sub-matrix of the first matrix. For the ith region, its equation of motion is that associated with the ith shaded region in the first matrix. That equation of motion is

$$[M_i] \{\ddot{q}_i\} + [C_i] \{\dot{q}_i\} + [C_n] \{\dot{q}_n\} + [K_i] \{q_i\} + [K_n] \{q_n\} =$$

$$\{f_i(t)\} \tag{6}$$

where $\{q_i\}$ is the vector for the coordinates of the ith region, and where $\{q_n\}$ is the vector for the coordinates of the regions which are neighbors to the ith region. The matrix $[M_i]$ is that for the ith shaded region of the first (the mass) matrix. The corresponding portions of the second (the damping) and the third (the stiffness) matrices are denoted as $[C_i]$ and $[K_i]$, respectively, and couple the equations for the ith region to those for the neighboring regions. It is noted that matrices $[M_i]$, $[C_i]$, and $[K_i]$ are square while the matrices $[C_n]$ and $[K_n]$ are not square but have the same number of rows as do the i matrices. Also, the vector $\{f(t)\}$ has the same number of rows as do the i matrices.

Equation (6) can be put in first order form by defining "state" variables $\{Z\}$ such that

$$\left\{Z_i\right\} = \left\{\begin{array}{c} q_i \\ \hline \dot{q}_i \end{array}\right\} \quad , \quad \left\{Z_n\right\} = \left\{\begin{array}{c} q_n \\ \hline \dot{q}_n \end{array}\right\}$$

so that (6) becomes

$$\left[\begin{array}{c|c} I & 0 \\ \hline 0 & M_i \end{array}\right] \left\{\dot{Z}_i\right\} = \left\{\begin{array}{c} 0 \\ \hline f_i(t) \end{array}\right\} - \left[\begin{array}{c|c|c|c} 0 & -I & 0 & 0 \\ \hline K_i & C_i & K_n & C_n \end{array}\right] \left\{\begin{array}{c} Z_i \\ \hline Z_n \end{array}\right\} \tag{7}$$

where I is the identity matrix.

From (7) there results

$$\left\{\dot{Z}_i\right\} = \left[\begin{array}{c|c} I & \\ \hline & M_i^{-1} \end{array}\right] \left(\left\{\begin{array}{c} 0 \\ \hline f_i(t) \end{array}\right\} - \left[\begin{array}{c|c|c|c} 0 & -I & 0 & 0 \\ \hline K_i & C_i & K_n & C_n \end{array}\right] \left\{\begin{array}{c} Z_i \\ \hline Z_n \end{array}\right\}\right) \tag{8}$$

which is the first order equation that will be solved by the node processor assigned to region i in the simulation system. The computational requirements for that node processor are determined below from this equation.

The largest linear problem that can be contained in the processor region i is one in which the matrices $[M_i]$, $[C_i]$, $[K_i]$, $[K_n]$, and $[C_n]$ are full. Therefore, treatment of matrices having rank equivalent to the number of

generalized coordinates for region i can be required. Also, the matrix $[M_i]$ must be inverted at every evaluation of $\{\dot{Z}\}$ in Equation (8). As a result, it may be necessary to invert at each time step a matrix ($\{M_i\}$) having rank equivalent to the number of generalized coordinates for region i.

Number of Generalized Coordinates

The limit of 200 for the number of generalized coordinates for region i was chosen arbitrarily. This number determines the local data memory size and the maximum time required to solve Equation (8).

Given that $\{q_i\}$ can have 200 elements, the mass matrix for region i can have rank 200. From reference [1], approximate numbers for multiplications/divisions and for additions/subtractions to invert a 200 x 200 matrix are 200^3 and $200(199)^2$, respectively. Carrying out the indicated multiplications gives approximately 8×10^6 multiplications/divisions and 8×10^6 additions/subtractions.

It should be noted that other computations are necessary in this linear problem besides the matrix inversion. Numerous matrix multiplications are also to be made, primarily those of a vector by a matrix. The numbers for multiplication/divisions and for additions/subtractions required in multiplying by an m x p matrix by an n x m matrix are $n \cdot m \cdot p$. The multiplication of a 200 x 1 matrix (a column vector) by a 200 x 200 matrix therefore requires about $200^2 = 4 \times 10^4$ multiplications/divisions and 4×10^4 additions/subtractions. This is negligible in comparison to the number of operations associated with the inversion of the mass matrix. Consequently, the inversion requirements are used to establish the overall multiplication/division and addition/subtraction requirements for the maximum linear problem to be treated by the processor for region i.

Externally Communicated Coordinates

The maximum coordinates for external communication from region i are 25% of the generalized coordinates used for region i.

The ratio of external to local coordinates can vary considerably with the specific problem being considered. A problem, consisting of two complex regions, connected by a few simple springs will be associated with a low ratio for each region. A problem consisting of regions intimately connected at many points will be associated with a large ratio for each region. For readily envisioned problems, the 25% ratio appears reasonable and one which need rarely be exceeded.

With this assumption, a maximum of 25% of the state variables for region i need be communicated to neighboring regions. Since there are 400 state variables maximum, 100 variables can be communicated to (and from) neighboring processors. As a result, the maximum size of each matrix $[C_n]$ and $[K_n]$ is 200 x 50.

With the size of $[C_n]$ and $[K_n]$ established, the sizes of all the matrices in Equation (8) for region i are known. These sizes are:

$[M_i]$ = 200 x 200

$[K_i]$ = 200 x 200

$[C_i]$ = 200 x 200

$[C_n]$ = 200 x 50

$[K_n]$ = 200 x 50

$\{f_i(t)\}$ = 200 x 1

$\{Z_i\}$ = 400 x 1

$[Z_n]$ = 100 x 1

$\{\dot{Z}_i\}$ = 400 x 1

It should be noted that this memory space for the largest linear problem does not include memory for intermediate results involved in matrix manipulations (e.g., inversion and multiplication), nor does it include program memory.

Nearest Neighbor Communications

A maximum of 6 neighboring processors will exchange state variable information with the processor for region i.

The number of neighboring processors was arbitrarily chosen, and corresponds to the physical elements which can apply forces or moments to the physical element represented by the processor for region i. Six is sufficiently large to allow a complex structure, but also the smallest number which can be used to treat a three-dimensional structure in a symmetric manner. It is noted that as many as 100 variables can be transmitted along any of these paths (if no variables are transmitted along the other 5 paths). If all 6 paths are transmitting information and if the number of variables transmitted on each path is the same, then each path will carry 100/6 variables.

Summary of Node Processor Requirements

Maximum computations per derivative equation (Equation (8)) per processor	
Multiplications/Divisions	8×10^6
Additions/Subtractions	8×10^6
Number of generalized coordinates per processor	200
Number of state variables per processor	400
Approximate numerical memory size (in floating point numbers) per processor (not including that for matrix manipulation)	150,000
Number of state variables transmitted to and from each processor	100

Number of separate transmission paths from each processor to neighbor	6
Primary desirable functions	matrix operations sin x $\tan^{-1} x$ $\sqrt{x}$
Secondary desirable functions	α^x log x

SOFTWARE REQUIREMENTS

The Digital System for Structural Dynamic Simulation, as can be seen by the previous discussion, has the requirement for a great variety of software. This software can be broken into five segments.

1. Offline programs for the central minicomputer to process user specifications of models, to prepare Node Processor programs for execution, and to aid in the development of Node Processor operational programs.
2. Realtime programs for the central minicomputer to load Node Processor programs and data, to coordinate execution of the model and to provide operator controls and displays.
3. Operational programs for the Node Processors to cause them to simulate substructures of the model and to communicate with neighbors and the central minicomputer.
4. Microcode (sometimes called firmware) to define the instruction set of the Node Processors.
5. Diagnostic Software.

HARDWARE REQUIREMENTS

Hardware requirements for the simulation hardware are listed below.

1. An array of Node Processors to perform the simulation.
2. Node Processors properties include:
   a) A large local memory for program and data storage.
   b) Fast instruction times.
   c) Auxiliary hardware for fast floating point multiplication, addition, subtraction, and division.
   d) Microcoded processor for a custom instruction set.
   e) Parallel communication paths to the six nearest neighbor Node Processors.
3. A central control minicomputer to direct the operation of the Node Processors and develop all associated software.

OVERVIEW OF SYSTEM ARCHITECTURE - SOFTWARE

The Digital System for Structural Dynamics Simulation consists of a three dimensional array of processors controlled by a minicomputer. Each processor in the array is capable of communicating directly with its six adjacent processors and with the minicomputer controller. The term Node Processor is used to identify a processor in the array.

The individual Node Processors are logically organized as a three dimensional array with each node having an address specified by three subscripts [e.g.: P(5,2,3)]. Each Node Processor will have a bidirectional communications link with the adjacent processor in both directions for each dimension. The first and last Node Processor in each dimension will be linked to make each dimension a full circle of processors (a hypertorous). The system has been designed with five processors in each dimension for a total of 125 processors. Processors will be numbered from 1 to 5 in each dimension. For example, P(2,5,3) will communicate directly with processors P(1,5,3), P(3,5,3), P(2,4,3), P(2,1,3), P(2,5,2), P(2,5,4); see Figures 2 and 3.

All of the processors will be connected to the minicomputer based controller with a bidirectional data and control bus. Data being sent from the controller to the processors will be prefaced with address information. Only the Node Processor(s) which need the data will store it in its local memory. When several Node Processors must transmit data over the common bus, the controller will command one Node Processor at a time to put data on the bus for use by the controller and/or other processors (Figures 4 and 5).

CONTROLLER OPERATING SYSTEM SOFTWARE

The software for using the Digital System to perform structural dynamics simulations will consist of the following five segments.

1. Offline programs for the central minicomputer to process user specifications of models, prepare Node Processor programs for execution and aid in the development of Node Processor operational programs.

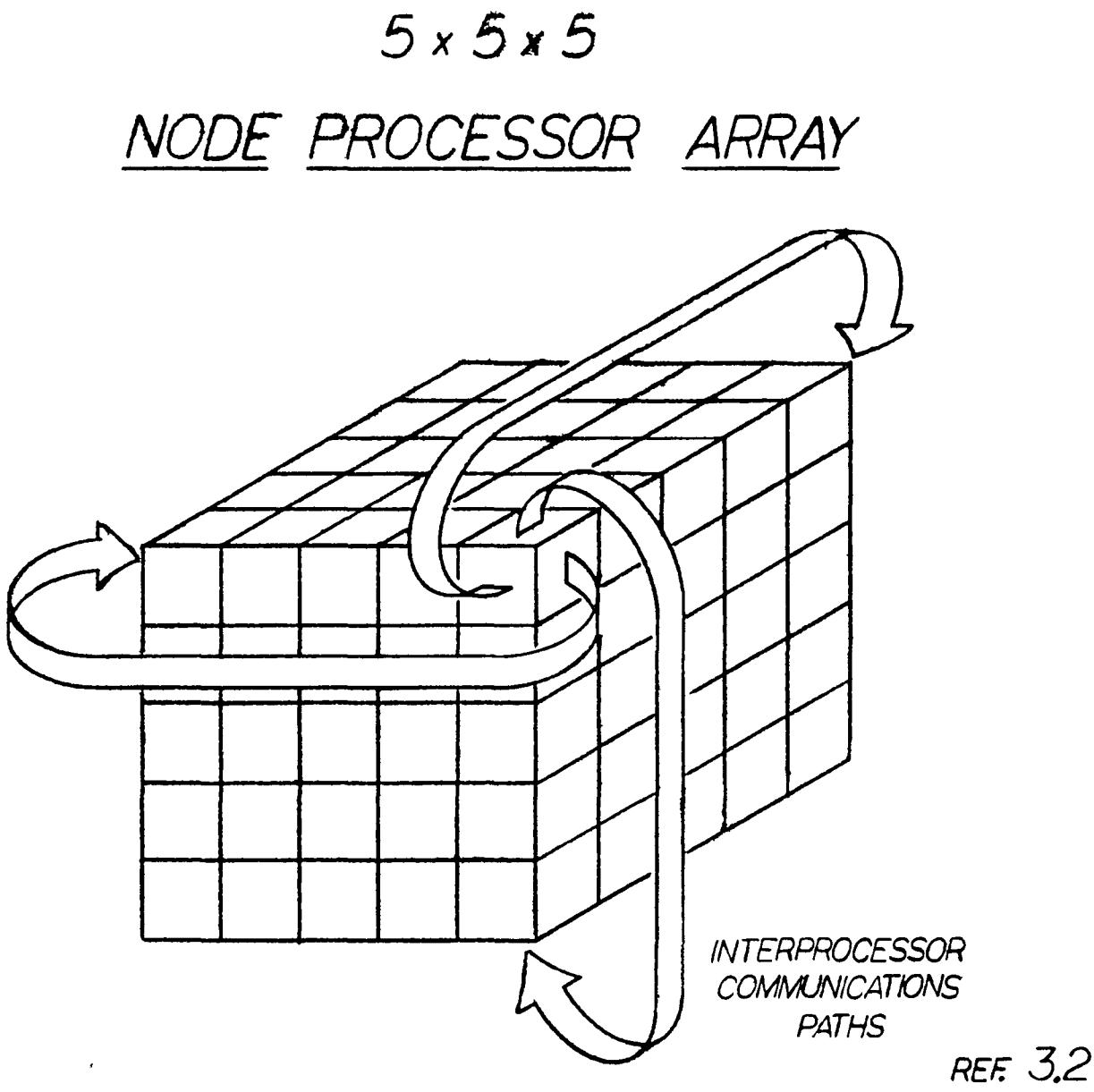


Figure 2 NASA/Lewis Structural Dynamics Simulator Block Diagram

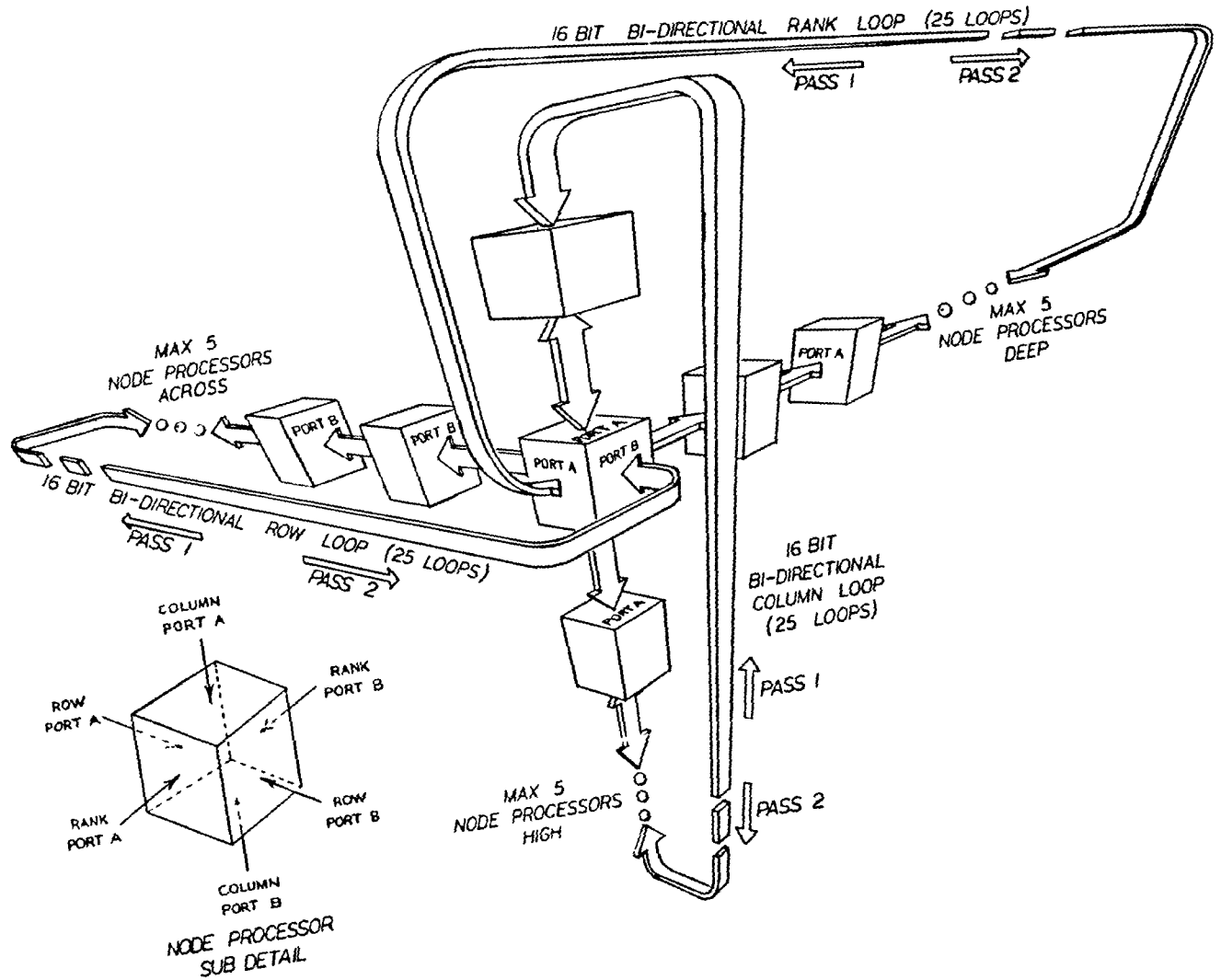


Figure 3 SDS To/From Six Nearest Block Diagram

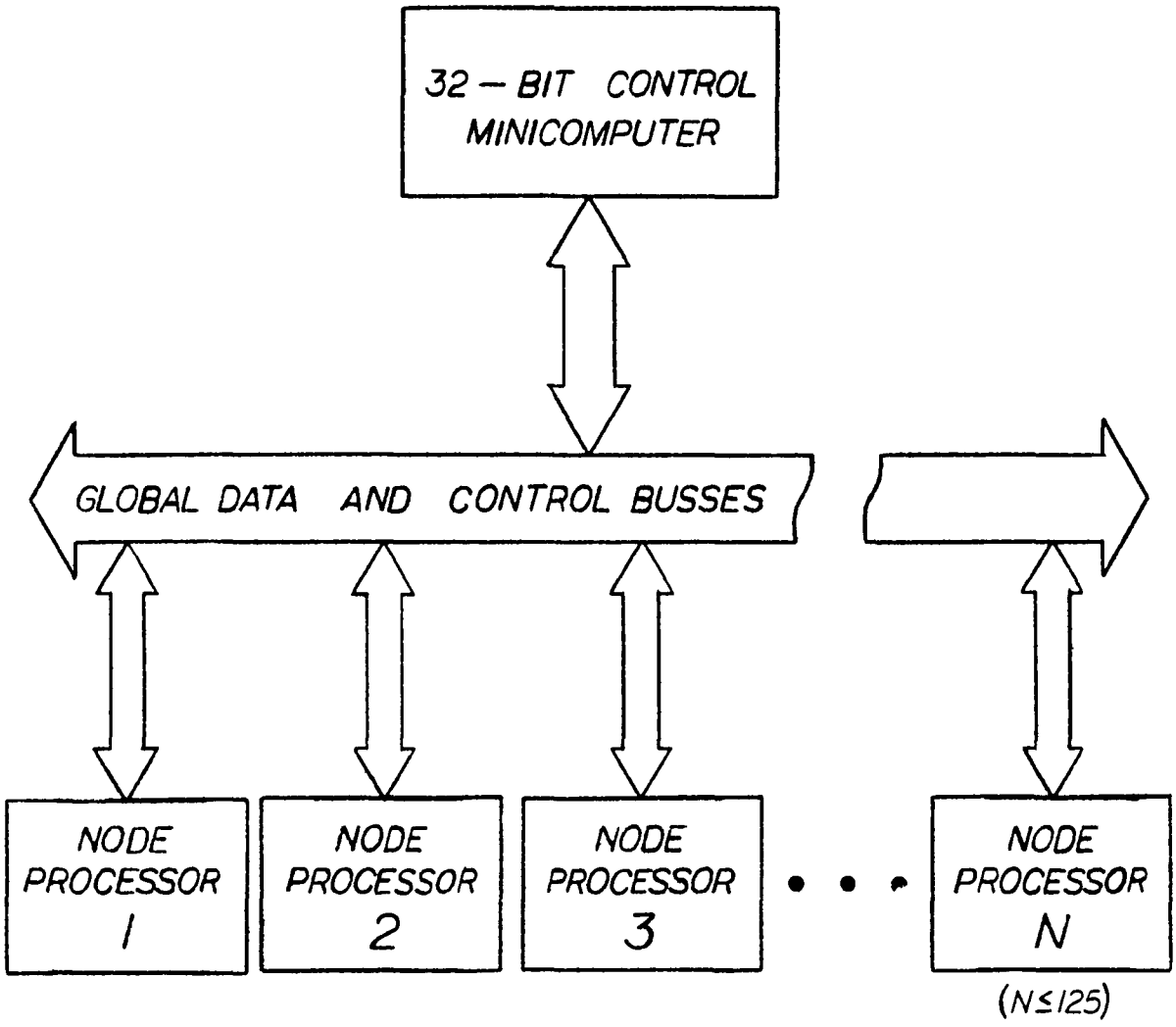


Figure 4 Structural Dynamics Simulator Block Diagram

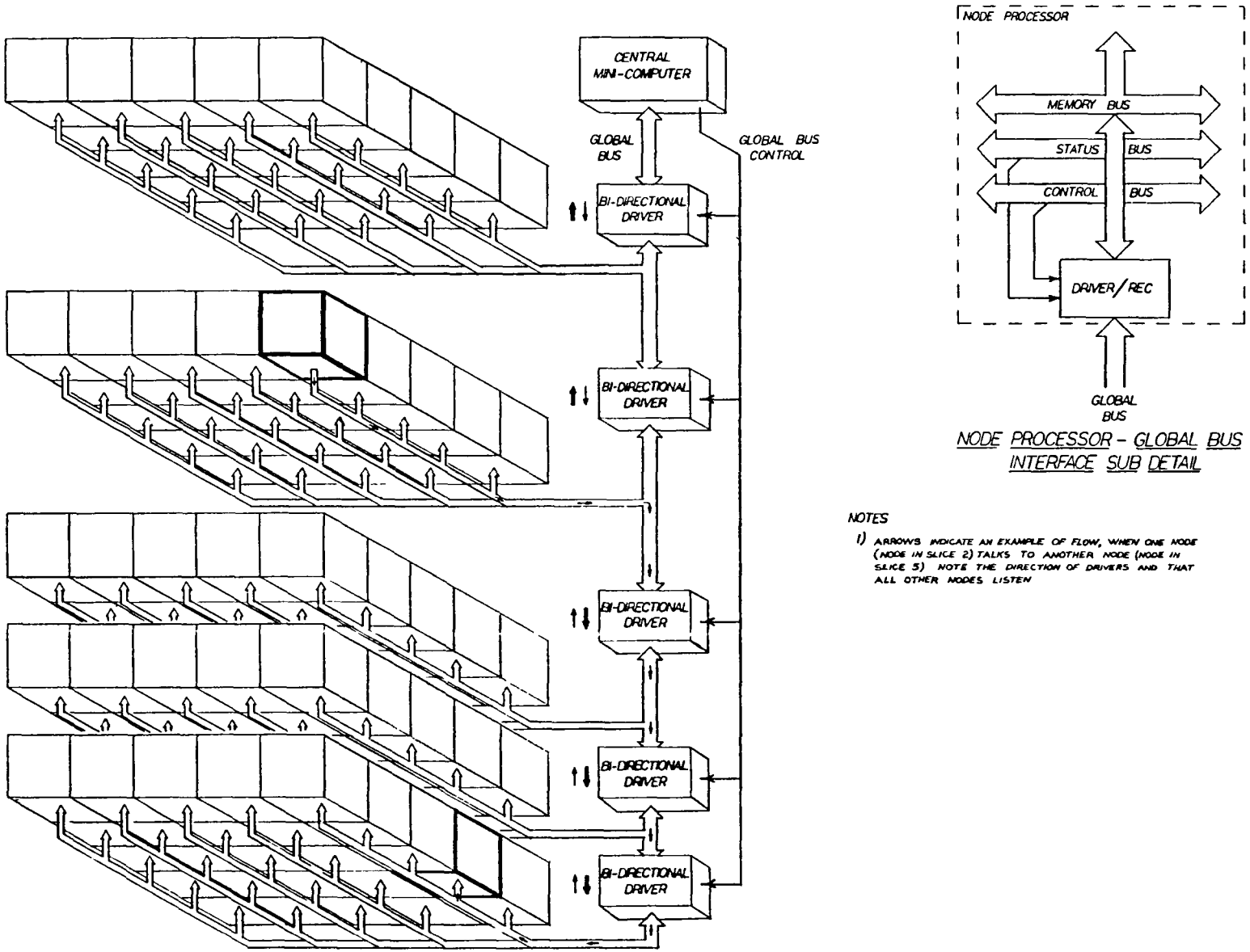


Figure 5 SDS Global Bus Interface Block Diagram

2. Realtime programs for the central minicomputer to load Node Processor programs and data, coordinate execution of the model and provide operator controls and displays.

3. Operational programs for the Node Processors to cause them to simulate substructures of the model and communicate with neighbors and the central minicomputer.

4. Microcode (sometimes called firmware) to define the instruction set of the Node Processors.

5. Diagnostic Software.

OFFLINE MODEL DEVELOPMENT SOFTWARE

The offline software will consist of a linker and assembler for generating the specific code to be executed by a Node Processor and a compiler for a Model Specification Language to assist users in programming the application software.

Node Processor Assembler

The Node Processor Assembler program will translate the symbolic assembly language programs written for the Node Processor into object modules that may be linked to other modules and form an executable memory image for the Node Processor. This assembler should have moderate macro capabilities and a full complement of assembler directives so as to ease the task of programming at the machine level. Also, the assembler should interface easily with the higher level Model Specification Language compiler.

Node Processor Linker

The Node Processor Linker will take one or more object modules created by the Node Processor Assembler and produce the memory image to be loaded into the Node Processor for execution. It is the linker's task to resolve all inter-module address references and relocate all intra-module addresses based on the location of the module within the executable memory image.

The linker should be capable of linking the user defined data tables created during model specification with the appropriate Node Processor operational programs. It is also the linker's task to build a symbol table for use by the Execution Control Program. The symbol table will be used when loading the Node Processors with data tables and functions when the operational program is already resident in the Node Processor memory. Also, the symbol table will be used during debugging and crash dump analysis.

Model Specification Language Compiler

A high order language will be developed which will consist of a set of rules governing the definition of models and forcing functions. The compiler program (sometimes called translator) will translate the model specification into the data tables and functions needed by the Node Processor operational programs.

The user will use the minicomputer text editor to create and modify text files containing the model specification. For linear problems, the bulk of the model specification will consist of numerical values for various matrices in the state equations. For non-linear problems, the user may need to specify a function for each element. These functions will be specified in Fortran-like arithmetic expressions. Other statements in the language will allow the user to specify the coupling between substructures, the forcing functions and the desired outputs.

The compiler program will read the text file containing the user's specification of the model and perform the following operations.

1. Assign substructures to processors using an algorithm to balance the processing load of each processor and optimize interprocessor communications. The compiler will be told how many Node Processors are operational in each dimension of the processor array. If there are more substructures than processors, substructures with the greater interaction will be assigned to the same processor.

To ensure proper balancing of the processing load the user will be able to specify the relative execution times for various substructures. The assignment subprogram will use the relative execution times as a weighting factor when assigning more than one substructure to a single processor. An iterative algorithm would be the simplest approach to making the best assignments.

Once the substructures have been combined, the compiler will optimize the interprocessor communications. Interacting substructures will be assigned to adjacent processors so that as much communication as possible can occur in parallel.

2. Generate the data tables each Node Processor operational program will use to solve the state equations for the substructures assigned to it.

3. Generate the tables to specify to each processor the inputs from each of its six adjacent processors, the inputs from the bidirectional data bus, the outputs to each of its six adjacent processors, and the outputs to the bidirectional data bus.

4. Generate a data file for the realtime central minicomputer program to specify the forcing function and the system outputs.

The compiler produces the required data modules which are loaded with the appropriate operational program into the Node Processor for execution. This makes the structure of the operational programs dependent on the output of the Model Specification Language compiler, but makes the compiler somewhat machine independent. Only the functions generated will be actual machine code. These functions will be specified in Fortran-like statements, thus making the language syntax independent of the Node Processor. The compiler's code generating functions will be tailored to the Node Processor.

The instruction set of the Node Processor was designed to assist the task of automatically generating code from Fortran arithmetic expressions. The data format for floating point numbers was chosen to conform to the popular PDP-11 data format. To simplify the generation of the data tables as described above, many addressing modes are available to the Node Processor for building address lists as needed.

REALTIME MODEL EXECUTION SOFTWARE

The realtime software will consist of a debugging facility for developing operational programs and an execution control program for doing actual simulations.

Execution Control Program

The Execution Control Program will be responsible for loading the Node Processors with operational programs, controlling the network of processors during execution, collecting data for animated displays as the execution proceeds and gathering the final results when the simulation completes.

Loading Node Processors

There will be two types of program loading for the Node Processors: The first type consists of loading the entire Node Processor with its operational program and data. The second type consists of only loading the data portions of the operational program. Typically, when the array of processors is brought up, they will be loaded with their operational programs and data for the first simulation run. Subsequent simulations will only be loaded into the data tables required by the operational programs if the model is compatible with the resident programs.

During the initial program load the Node Processor will be forced to execute its microcode from location 0. The microcode at this location may execute a few diagnostics to determine the integrity of its local memory and exercise the communications link with the controller. The processor will then wait for a command from the central minicomputer controller. The central mini-

computer will poll the Node Processors to determine which are on line and ready.

When each Node Processor has acknowledged its readiness status, the central minicomputer controller will send a command to a processor telling it to accept the ensuing data stream as a memory image. Following the memory image will be the initial program counter value. Typically, this value will address a program routine which will wait for the "Start Execution" command from the controller.

After the execution of a simulation is completed, the operational program on the Node Processor will enter a routine to wait for additional commands from the controller. Subsequent simulations which use the same operational program will then need only to load the data portion of the programs. The operational programs should be table driven as much as possible to accommodate this scheme.

Controlling the Processor Network during Execution

The Execution Control Program will broadcast the start simulation command to all nodes. The operational programs on the Node Processors will then execute the time step until it needs to communicate with other processors. A control bus line will be raised by the Node Processor when it is ready. The bus line will only appear active to the controller once all processors are ready. The controller will then send the commands to initiate transfers between neighbors. When transfers are completed in the requested direction, the controller is then notified by each processor. After all are ready again, the controller will broadcast to the nodes to start transferring data in the opposite direction.

After the nearest neighbor communications have completed, the controller will command each node in turn to put its global state variables on the data bus. All processors will listen to the global bus as state variables appear and will store only those state variables from other nodes as it is directed by its internal tables. The controller will also store those state variables it needs for the graphics display.

Once all the state variables have been transferred, the controller may then interrogate individual Node Processors for additional data needed to update the graphics display. When Node Processors have completed all the necessary communications, they will continue on with their simulation computations.

During the communications procedure all on-line Node Processors are required to signal acceptance of data. If the controller does not receive the acceptance signal, the controller will time out. The Execution Control Program will interrogate the Node Processors individually to determine which one(s) are not responding. Special status lines on the control bus may be used to selectively put a Node Processor offline or cause the processor to go into a microcoded diagnostic routine to determine the reason for the failure.

Collecting Graphics Display Data

The Execution Control Program will interrogate selected Node Processors for data needed to drive a graphics display. The collection of the data will be directed by the data tables generated during the Model Specification compilation. The data transfers from Node Processors to central minicomputer will take place between time steps of the simulation.

Gathering Final Results

The Execution Control Program will collect the final state variables from each Node Processor after the simulation has completed. The data collected will be stored in disk files for analysis by other analysis programs. The data tables generated during compilation of the Model Specification will direct the Execution Control Program to command the appropriate variables to be sent from a node to the host.

Node Processor Program Debugger

A Node Processor Program Debugger facility is needed to assist in the development of the operational programs. This debugger will perform the functions required by the realtime Model Execution Program but will support more operator interaction.

A special library of debugger modules will be available to the program developer. These modules will be linked to operational programs under development to support features such as instruction execution tracing, breakpoints and memory examination/modification.

When programs are linked to the debugger modules and loaded into the processors with the Node Processor Debugger Facility, the operator may interact with the execution of the program. This powerful facility will greatly increase the system programmer's productivity.

NODE PROCESSOR OPERATIONAL PROGRAM

The Node Processor Operational Programs are those programs which will cause each processor to simulate a substructure and communicate with its neighbor and the central minicomputer. The programs will be written in assembly language or possibly a Fortran-like language that is easily translated into assembly language code.

The content of these programs will be highly dependent on the actual model being used for the simulation. The programs should be structured so as to operate on the data tables that are generated by the Model Specification compiler. The data tables and special functions generated by the compiler must be easily linked with the Operational Program to perform the required simulation.

NODE PROCESSOR MICROCODE

The Node Processors will be microcoded to provide a typical general purpose minicomputer instruction set. In addition to the general purpose instruction set, special high-level microcoded routines will be available for doing matrix and vector operations, communicating with nearest neighbors and the central minicomputer and providing support for diagnostics. The microcoded instruction set is described in the Node Processor Instruction Set Reference.

DIAGNOSTIC SOFTWARE

Diagnostic software will be implemented at the following three levels.

1. Diagnostic Control Program in the central microcomputer.
2. Diagnostic Program in each processor.
3. Diagnostic Machine instructions implemented in microcode.

The Diagnostic Control Program in the central minicomputer will load the Diagnostic Program into each processor, accept operator direction, issue control commands to the processors, input results from the processors, and display the results to the operator.

The Diagnostic Program in each processor will run various tests as commanded by the central minicomputer. The tests will progress from simple ones which test a minimum of circuitry to more complex ones. Tests will be included for the interprocessor communications, bidirectional bus communications, and the internal circuitry on each microprocessor board.

The Diagnostic Machine instructions will be designed to thoroughly exercise the processor board circuits and aid in fault isolation.

NODE PROCESSOR ARCHITECTURE

The Node Processor is part of an array of processors, each capable of communicating with a central host computer and directly with six neighbors. It consists of a bit-slice 32-bit CPU, up to 256K of 64-bit memory, 256 72-bit floating point scratch pad registers and floating point units capable of overlapping multiplication with either addition, subtraction or division. The instruction set of the Node Processor includes a full complement of instructions typically found on a general purpose minicomputer plus additional high level instructions to handle vectors and matrices.

Each instruction for the Node Processor is 64 bits and contains an opcode and up to 2 operand fields. There are 13 possible addressing modes per

operand with a minimum of restrictions on the addressing modes available for each instruction. With over 130 opcodes and up to 13 addressing modes per operand there are over 10,000 distinct instructions defined. This addressing scheme gives the Node Processor a very versatile and powerful instruction set.

MEMORY

Main memory appears as 64 bits to the machine level programmer. The internal data bus connecting memory with the bit-slice CPU and floating point scratch pad memory is 32 bits wide, therefore 2 transfers are made to access a 64-bit word and only 1 transfer for a 32-bit half-word. The number of memory transfers required by a processor instruction is dependent on the type of data being manipulated. The actual transfers made are under control of the microcode and are not the responsibility of the machine language programmer. The machine language programmer is capable of addressing memory only at 64-bit word boundaries, thus there is never the possibility of addressing memory at a half word boundary. If memory were addressed at 32-bit boundaries under program control, it is possible that alignment problems could arise when addressing 64-bit floating point quantities. This problem is eliminated since all programs are restricted to addressing at 64 bit boundaries.

The memory is 1-bit error correcting, 2-bit error detecting. If the Node Processor detects a 2-bit error during execution of an instruction a trap via location 4 is performed. One bit errors are corrected by the hardware and the currently executing instruction is completed normally.

There is a memory address comparison register (MACR) which is accessed whenever memory is written. When memory is written the address being used is compared to the value in MACR and the appropriate bits in the processor status word (PSW) are set.

Main memory is used to hold data and program instructions. With the exception of the lowest 70 words of memory, instructions and data may be anywhere in memory. Memory locations 0 through 69 are used for processor traps and

holding constants required by certain high level instructions. Main memory size is either 256K, 512K, 768K or 1 million 32-bit words. Each memory board in the system may hold 256K 32-bit words (128K 64-bit words).

DATA TYPES

There are 4 data types handled by the Node Processor: integer, floating point, vectors and matrices. Integer data is 32 bits wide and left justified in a 64-bit memory word. The LSB of the integer is at bit 32 while the MSB is at bit 63. The integer data type is used to represent numerical quantities and hold memory addresses. When used as an address, only the low order 19 bits of the integer are used. However, all 32 bits are used for calculating memory addresses and no checks are made to ensure that the unused portion of the integer is all zeroes.

Floating point data is 64 bits in main memory and 72 bits in the floating point scratch pad memory. The increased scratch pad representation allows for 7 extra bits of precision in the mantissa. When transferring main memory to the scratch pad registers, the 64-bit number must be expanded to 72-bits. The sign and exponents will be the same in both formats. Bit 62 is set to 1 in the scratch pad register if the exponent is non-zero. The main memory mantissa is mapped to bits 61-7 of the scratch pad register and bits 6-0 are set to 0 to complete the mantissa. When going from the scratch pad registers to main memory, the sign and exponent are copied directly and bits 61-7 are taken as the mantissa. (Note: The MSB bit of the mantissa, 1 if non-zero number, is not stored in main memory. The mantissa is truncated to 55 bits when going to main memory.)

Vectors are arrays of floating point values which reside in consecutive locations of main memory. The address of a vector is the location of the first floating point value. The length of a vector is the number of floating point values making up the vector.

Matrices are two dimensional arrays of floating point values and reside in a contiguous block of main memory. Each row of the matrix is stored in consecutive memory locations with the first row starting at the beginning

address of the matrix. If there are N elements in a row then the second row will start at the matrix address + N. Likewise, the third starts at matrix address + 2N and so on until the number of rows in the matrix is exhausted. The elements of each column in a matrix are separated by N-1 memory locations. (Note: The autoincrement addressing modes of the instruction set enable the programmer to easily access each memory of a column without the need for doing complex address calculations.)

NODE PROCESSOR REGISTERS

The machine level programmer has access to 8 general purpose 32-bit integer registers, 3 special purpose 32-bit integer registers and 250 72-bit floating point scratch pad registers. The general purpose integer registers are designated R0 through R7 and are used to hold integer numerical quantitives and addresses. These registers are not affected by the Node Processor instruction unless specified as an operand by the programmer.

The special purpose integer registers are the system stack pointer (SP), program counter (PC) and a vector/matrix size register (MS). The stack pointer contains the address of an area of memory set aside by programs for dynamic storage. The stack grows (increasing memory addresses) as words are placed (pushed) onto it and shrinks (decreasing memory addresses) as words are removed (popped). The program counter is used to specify the location from which the next instruction is to be taken. The vector/matrix size register stores an integer value which conveys the length of a vector or the number of rows of matrix to certain high level instructions.

The floating point scratch pad registers are designated F0 through F249. They are general purpose floating point accumulators and are under programmer control except for the actions of certain high level instructions which may destroy their contents during execution.

PROCESSOR STATUS WORD

The Processor Status Word (PSW) contains information on the current status of the Node Processor. Selected portions of the PSW are affected by the

execution of instructions. Each instruction described below will detail which bits of the PSW are affected during execution. In general, bits 0-3 reflect the results of the last 32-bit integer operation on the bit-slice CPU, bits 4-7 reflect the results from the last floating point multiply operations, bits 8-11 reflect the last floating point add/subtract/divide result, bits 15-16 reflect the error status of the last memory read, bits 13-14 are under the programmers control for enabling traps, bit 17 is concerned with host/node synchronization and bits 18-20 reflect comparisons of memory addresses when doing writes.

High level instructions affect the PSW in a manner different from simple instructions. Since all high level instructions involve multiple operations on either or both of the floating point units, the final PSW state reflects multiple operations. The only status bits which are meaningful are the overflow (FOV1,FOV2) and divide by zero (FDZ) bits. If any of these bits are set during execution of a high level instruction then they will be set at the end of instruction. The status of all other PSW bits for the integer and floating point units are undetermined.

Bit	Mnemonic	Contents
0	CR	Integer carry
1	OV	Integer overflow
2	NE	Integer result negative
3	ZE	Integer result zero
4	FUN1	FP multiply underflow
5	FOV1	FP multiply overflow
6	FNE1	FP multiply negative
7	FZE1	FP multiply result zero
8	FUN2	FP add/subtract/divide underflow
9	FOV2	FP add/subtract/divide overflow
10	FNE2	FP add/subtract/divide result negative
11	FZE2	FP add/subtract/divide result zero
12	FDZ	FP divide by zero
13	FTR	Trap enable on FDZ, FOV1, or FOV2 via location 10

14	TRC	Trace enable. Trap after each instruction.
15	PAR	One bit error from memory access
16	RER	Two bit error from memory access
17	RDY	Write only. Signals to host this processor is done with current step.
18	BPT	Enable trap on setting of FEN
19	FEN	Memory write address equals MACR
20	FLT	Memory write address less than MACR

INPUT/OUTPUT

The Node Processor communicates with the central host computer and each of its six nearest neighbors. I/O is synchronous in that there are no interrupts caused by I/O transfers. Each Node Processor program must explicitly invoke an I/O command before any data transfer.

PROCESSOR TRAPS

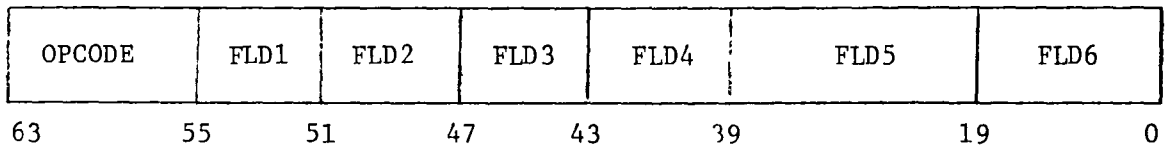
There are a number of exceptional conditions which cause the Node Processor to trap to fixed locations. These conditions may be due to hardware or software failures. When a trap is executed the error condition will cause the processor to push the present PSW and PC onto the system stack and take the new PSW and PC from consecutive memory locations. The memory locations and conditions are listed below. It is the software's responsibility to load the proper addresses for the trap service routines.

Trap Condition	Location
Two-bit Memory Failure	4
Illegal Instruction	6
Out of Limits (CLIM)	8
FTR and (FOV1 or FOV2 or FDZ)	10
Attempt SQRT with Negative	12
Stack Underflow	14
Trace Enabled	16
FEN and BPT	18

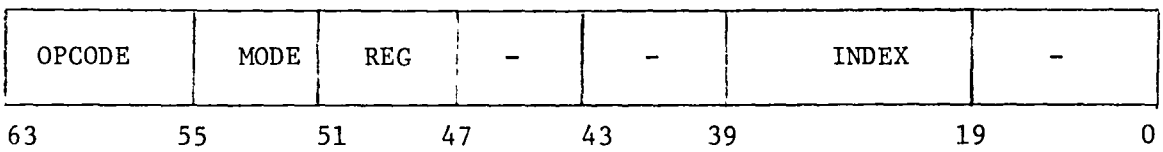
INSTRUCTION FORMATS

The Node Processor instruction length is 64 bits. Each instruction contains an 8-bit opcode field which allows up to 256 distinct opcodes. Since less than 256 instructions have been defined, there are opcodes left for future instructions. Instructions may require zero or more operands for execution. For instructions with 2 or less operands, each operand is completely specified within the instruction word. A 4-bit address mode, 4-bit register select and a 20-bit field are contained in the instruction to specify the effective address of the operand. The 20-bit quantity may represent an index, immediate operand, actual address, floating point register select, general register to autoincrement by or autoincrement value as required by the addressing mode. In the description of instruction formats below, this 20-bit quantity is always labeled index though it may represent something else depending on the addressing mode.

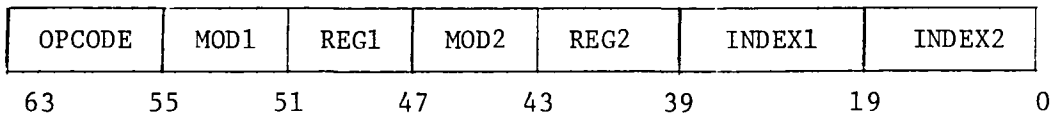
For instructions with 3 or more operands the first 2 operands will be specified as in the 2-operand instruction while the rest of the operands will be taken from pre-specified registers as defined by each instruction. The general format of the 64-bit instruction is:



For no operand instructions only the 8-bit opcode field is used. For 1-operand instructions FLD1 specifies the operand addressing mode and FLD2 specifies the general register to be used if required by the addressing mode. If the addressing mode requires an index, displacement, actual address or autoincrement value then it is contained in FLD5. The general format for a 1-operand instruction is:



For 2-operand instructions, the first operand is specified as in the 1-operand instruction. The second operand uses FLD3, FLD4, FLD6 to specify addressing mode, register select and index respectively. The format of a 2-operand instruction is:



The uses of the fixed fields in the 64-bit instruction are outlined below.

Instruction

Field	Bits	Contents
OPCODE	63-56	Operation code. (0-255)
FLD1	55-52	First address mode. (0-12)
FLD2	61-48	Register select. 0-7 for R0 through R7. 8 for MS, 9 for SP, 15 for PC (Note: 10-14 not allowed. These registers are dedicated to the microcode.)
FLD3	41-44	Second address mode (0-12)
FLD4	43-40	Second register select as in FLD2
FLD5	39-20	First address index value, immediate operand, actual address, register select (as in FLD2) with auto-increment value or F.P. register select (0-249)
FLD6	19-0	Second address index value, immediate operand, actual address, register select with autoincrement value or F.P. register select (0-249).

NODE PROCESSOR INSTRUCTIONS

OPCODE	OPERANDS	DESCRIPTION
Integer Instructions:		
MOVE	SRC,DST	Move integers
BLKM	ADR1,ADR2	Move block of integers
ADD	SRC,DST	Add integers
SUB	SRC,DST	Subtract integers
MUL	SRC,DST	Multiply integers
DIV	SRC,DST	Divide integers
JSB	Rn,ADR	Jump to subroutine
RSB	Rn	Return from subroutine
PUSH	SRC	Push onto stack
POP	SRC	Pop off stack
SAVEM	Rn,CNT,ADR	Save multiple registers
LOADM	ADR,Rn,CNT	Load multiple registers
JMP	ADR	Unconditional jump
JMPLE	ADR	Conditional jump if <=
JMPLT	ADR	Conditional jump if <
JMPGE	ADR	Conditional jump if >=
JMPGT	ADR	Conditional jump if >
JMPEQ	ADR	Conditional jump if =
JMPNE	ADR	Conditional jump if <>
SKPLE	SRC1,SRC2	Compare and skip if SRC1 <= SRC2
SKPLT	SRC1,SRC2	Compare and skip if SRC1 < SRC2
SKPGE	SRC1,SRC2	Compare and skip if SRC1 >= SRC2
SKPGT	SRC1,SRC2	Compare and skip if SRC1 > SRC2
SKPEQ	SRC1,SRC2	Compare and skip if SRC1 = SRC2
SKPNE	SRC1,SRC2	Compare and skip if SRC1 <> SRC2
TSTLE	SRC,ADR	Test and jump if <= 0
TSTLT	SRC,ADR	Test and jump if < 0
TSTGE	SRC,ADR	Test and jump if >= 0
TSTGT	SRC,ADR	Test and jump if > 0
TSTEQ	SRC,ADR	Test and jump if = 0
TSTNE	SRC,ADR	Test and jump if <> 0
BITNE	SRC1,SRC2	Test bit(s) and skip if not zero
BITEQ	SRC1,SRC2	Test bit(s) and skip if zero
SJGT	SRC,ADR	Subtract and jump if positive
TRAP	ADR	Trap
RTP		Return from trap
SPS	SRC	Set processor bits
CPS	SRC	Clear processor bits
RPS	DST	Read processor bits
TPSNE	SRC	Read PSW and skip if selected bits are set
TPSEQ	SRC	Test PSW bits and skip if 0
SWAP	SRC,DST	Swap words
ASHL	CNT,DST	Arithmetic shift left

ASHR	CNT,DST	Arithmetic shift right
LSHL	CNT,DST	Logical shift left
LSHR	CNT,DST	Logical shift right
AND	SRC,DST	Logical and
IOR	SRC,DST	Inclusive or
XOR	SRC,DST	Exclusive or
COM	SRC,DST	Complement
NEG	SRC,DST	Negate
CLR	DST	Clear
INC	DST	Increment
DEC	DST	Decrement
ADC	DST	Add carry
ABS	SRC,DST	Absolute value
CEA	SRC,DST	Compute effective address
CLIM	LO,HI	Compare against limits
CASE	SRC,HI	Case statement
ISKP	SRC,DST	Increment and skip if within limit
DSKP	SRC,DST	Decrement and skip if within limit

Floating Point Instructions:

FADD	Fm,Fn	Floating add, register to register
FADD	Fn,DST	Floating add, register to memory
FADD	SRC,Fn	Floating add, memory to register
FADD	SRC,DST	Floating add, memory to memory
FSUB	Fm,Fn	Floating subtract, register to register
FSUB	Fn,DST	Floating subtract, register to memory
FSUB	SRC,Fn	Floating subtract, memory to register
FSUB	SRC,DST	Floating subtract, memory to memory
FMUL	Fm,Fn	Floating multiply, register to register
FMUL	Fn,DST	Floating multiply, register to memory
FMUL	SRC,Fn	Floating multiply, memory to register
FMUL	SRC,DST	Floating multiply, memory to memory
FDIV	Fm,Fn	Floating divide, register to register
FDIV	Fn,DST	Floating divide, register to memory
FDIV	SRC,Fn	Floating divide, memory to register
FDIV	SRC,DST	Floating divide, memory to memory
FMOV	Fm,Fn	Floating move, register to register
FMOV	Fn,DST	Floating move, register to memory
FMOV	SRC,Fn	Floating move, memory to register
FMOV	SRC,DST	Floating move, memory to memory
FBLK	ADR1,ADR2	Floating block move
FPUSH	SRC	Push floating
FPOP	DST	Pop floating
FSAVE	Fn,CNT,ADR	Save floating point registers Fn to F<n+CNT-1>
FLOAD	ADR,Fn,CNT	Load Floating point registers from memory
FSKPLE	SRC1,SRC2	Compare floating and skip if SRC1 <= SRC2
FSKPLT	SRC1,SRC2	Compare floating and skip if SRC < SRC2
FSKPGE	SRC1,SRC2	Compare floating and skip if SRC1 >= SRC2
FSKPGT	SRC1,SRC2	Compare floating and skip if SRC1 > SRC2
FSKPEQ	SRC1,SRC2	Compare floating and skip if SRC1 = SRC2
FSKPNE	SRC1,SRC2	Compare floating and skip if SRC1 <> SRC2

FJMLE	ADR	Branch if FP multiply result <= 0
FJMLT	ADR	Branch if FP multiply result < 0
FJMGE	ADR	Branch if FP multiply result >= 0
FJMGT	ADR	Branch if FP multiply result > 0
FJMEQ	ADR	Branch if FP multiply result = 0
FJMNE	ADR	Branch if FP multiply result <> 0
FJLE	ADR	Branch if FP add, divide or subtract result <= 0
FJLT	ADR	Branch if FP add, divide or subtract result < 0
FJGE	ADR	Branch if FP add, divide or subtract result >= 0
FJGT	ADR	Branch if FP add, divide or subtract result > 0
FJEQ	ADR	Branch if FP add, divide or subtract result = 0
FJNE	ADR	Branch if FP add, divide or subtract result <> 0
FTSTLE	SRC,ADR	Branch if floating SRC <= 0
FTSTLT	SRC,ADR	Branch if floating SRC < 0
FTSTGE	SRC,ADR	Branch if floating SRC >= 0
FTSTGT	SRC,ADR	Branch if floating SRC > 0
FTSTEQ	SRC,ADR	Branch if floating SRC = 0
FTSTNE	SRC,ADR	Branch if floating SRC <> 0
FABS	SRC,DST	Absolute value of floating point

High Level Instructions:

SIN	SRC,DST	Sine
COS	SRC,DST	Cosine
ATAN	SRC,DST	Arc tangent
SQRT	SRC,DST	Square Root
MTMUL	MAT1,MAT2	Matrix multiplication
MTADD	MAT1,MAT2	Matrix addition
MVMUL	MAT,VEC	Matrix-times vector
SOLVE	MAT,VEC	Gaussian Elimination
MSVEC	SRC,DST	Scalar times vector
ASVEC	SRC,DST	Add scalar to vector
VCMUL	SRC1,SRC2	Cross product
VCADD	SRC1,SRC2	Add 2 vectors
MTMUL2	MAT1,MAT2	Generalized matrix multiplication general
MTADD2	MAT1,MAT2	Generalized matrix addition
MVMUL2	MAT,VEC	Generalized matrix times vector
RKM1	SRC1,SRC2	Performs step 1 of the integration
RKM2	SRC1,SRC2	Performs step 2 of the integration
RKM3	SRC1,SRC2	Performs step 3 of the integration
RKM4	SRC1,SRC2	Performs step 4 of the integration
RKM5	SRC1,SRC2	Performs step 5 of the integration
RKERR	SRC,DST	Computes the local truncation

TABLE 1

Addressing Mode Execution Times

Modes		Integer Source	F.P. Source	Integer Destination	F.P. Destination	ADR
0	Register	1	1	1	1	-
1		4.5	6	6.5	8	1
2	Immediate	1	1	-	-	-
3	Memory	4.5	6	6.5	8	1
4		4.5	6	7.5	9	2
5	Indirect	8	9.5	11	12.5	4.5
6		5.5	7	7.5	9	2
7		4.5	6	6.5	8	2
8	Indirect	8	9.5	12	13.5	5.5
9	Indirect	8	9.5	11	12.5	4.5
10	Indirect	8	9.5	11	12.5	4.5
11		5.5	7	7.5	9	2
12		4.5	6.0	6.5	8	2

The numbers are μcycles. If the μcycle involves a memory access, then it is 1.5 μcycles. Integer memory reference modes use 1 memory access, floating point involves 2. Indirect integer modes have 2 memory cycles, floating point indirect modes involve 3 memory accesses.

TABLE 2

Addressing Mode Time Estimates in μseconds
Uses Table 1 μcycles * 150 nsec. per cycle

Mode		Integer Source ISRC	F.P. Source FSRC	Integer Dest. IDST	F.P. Dest. FDST	Jump Address JADR
0	Register	.15	.15	.15	.15	-
1		.675	.9	.975	1.2	.15
2	Immediate	.15	.15	-	-	-
3	Memory	.675	.9	.975	1.2	.15
4		.675	.9	1.125	1.35	.3
5	Indirect	1.2	1.425	1.65	1.875	.675
6		.825	1.05	1.125	1.35	.30
7		.675	.9	.975	1.2	.30
8	Indirect	1.2	1.425	1.8	2.025	.825
9	Indirect	1.2	1.425	1.65	1.875	.675
10	Indirect	1.2	1.425	1.65	1.875	.675
11		.825	1.05	1.125	1.35	.30
12		.675	.9	.975	1.2	.30

Used 150 nsec for read cycle time; 225 nsec for memory read/write time.

(Usually there are 2 cycles involved in memory access; 1st does read, 2nd does check for 1 or 2 bit error. Would save overall time if 2nd μcycle was short.)

TABLE 3

Example Execution Times for Instruction
(NOTE: Each Instruction is Without Fetch Cycle
Which Adds in 4 μcycles (.6 μsec.))

Times in μcycles

MOVE		2 + ISRC + IDST
ADD		2 + ISRC + IDST
JMP		1 + JADR
		5 + JADR
FADD	R/R	6 + Add time
	R/M	10 + Add time + FDST
	M/R	6 + Add time + FSRC
	M/M	6 + Add time + FSRC + FDST
FMUL	R/R	6 + Mult. time
	R/M	10 + Mult. time + FDST
	M/R	6 + Mult. time + FSRC
	M/M	6 + Mult. time + FSRC + FDST
FMOV	R/R	6
	R/M	10 + FDST
	M/R	6 + FSRC
	M/M	6 + FDST + FSRC
SINE (typical)		73 + 14 adds + 11 multiplies
ATAN (typical)		76 + 10 adds + 6 multiplies + 2 divides
MTMUL (Matrix multiply)		$36 + R*(3 + C*\underbrace{(2 + \text{add time})}_{\text{overlaps multiply}})$
SOLVE (Gaussian Elimination)		on order of $2N*DT + N^2*MT + \frac{N^3}{2}*ST + N^2*ST + \frac{N}{2}*ST$
		where DT = divide time ST = subtract time MT = multiply time N = # of rows, # of columns in matrix
RKM1		9 + N*(4 + MT) + FSRC + FDST

TABLE 4

The following are the expected execution times for various Node Processor instructions. To simplify the tables and give a best guess sampling of instruction times, the following assumptions have been made.

a. For operands labled REGISTER, the operand may come from a register or be immediate (i.e., part of instruction).

b. Operands labeled MEMORY are either mode #1 or #3. (Probably the most typical.)

c. Average microcycle time for bit slice is 150 nsec., instruction with memory access is 1.5 microcycles. See Table 2 for explanation.

d. Floating point operation times used:

1.0 = Multiply
1.5 = Add or Subtract
15.4 = Divide

These numbers used would probably be smaller with sparsely filled matrices since operations with one zero operand are significantly faster.

e. Times do not include the instruction fetch times, which will be approximately .6 microseconds per instruction.

Instructions with comparable execution times :

MOVE, ADD, SUB, AND, IOR, XOR, COM, NEG, INC, DEC, ADC, ABS, PUSH, POP

JMP, JMPLE, JMLT, JMPGE, JMPGT, JMPEQ, JMPNE

SKPLE, SKPLT, SKPGE, SKPGT, SKPNE, SKPGE, BITNE, BITEQ

TSTLE, TSTLT, TSTGE, TSTGT, TSTEQ, TSTNE

ASHL, ASMR, LSHL, LSHR

FADD, FSUB

FMOV, FPUSH, FPOP

FSKPLE, LT, GT, GE, NE, EQ

FTSTLE, LT, GT, GE, NE, EQ

FJMLE, LT, GE, GT, NE, EQ, FJLE, LT, GE, GT, NE, EQ

Function	Register to Register	Register to Memory	Memory to Register	Memory to Memory
MOVE	.6	1.425	1.125	1.95
ADD	.6	1.425	1.125	1.95
JMP	-	-	-	.3
JSB	-	-	-	.9
FADD	2.4	4.2	3.3	4.5
FMUL	1.9	3.7	2.8	4.0
FMOV	.9	2.7	1.8	3.0
SINE	42.95	44.15	43.85	45.05
ATAN	63.2	64.4	64.1	65.3
MTMUL (10x10)	-	-	-	189.9
(50x50)				4.53 ms
SOLVE (10x10)	-	-	-	1.3 ms
(50x50)				101.6 ms
RKM1 (10) state variables				17.35
(50) state variables				81.35

* Time in μseconds except as noted

SOFTWARE SPECIFICATION SUMMARY

The software identified for the Digital System for Structural Dynamics Simulation includes the following segments.

1. Offline Model Development Software.
2. Realtime Execution Control Software.
3. Node Processor Operational Programs.
4. Node Processor Instruction Set.

Offline Model Development Software

The Offline Model Development Software includes the linker/assembler programs for generating Node Processor executable programs and a compiler to translate user specifications of simulation problems into a form that can be handled by the Node Processor operational programs.

The assembler and linker programs are necessary to generate programs for the Node Processor. These programs are needed for any newly designed computer system.

The Model Specification language compiler is designed to quicken the task of preparing for simulation runs. The language must first be designed before the compiler (translator) can be fully specified. Before the language is designed, a sample problem should be identified. The purpose of the sample problem is to provide a focal point during the development of the language. It will provide useful guidelines for the language and thus reduce the probability of a costly over-generalized language.

The translator should not be considered as a full blown compiler for languages such as Fortran, Basic and ADA. The language in all likelihood would be rigidly formatted and limited in scope so as to ease the tasks required by a compiler. The compiler would need to parse arithmetic expressions and generate the appropriate code for the Node Processor. Elaborate capabilities would probably not be cost effective.

Realtime Execution Control Software

The Realtime Execution Control Software includes a debugging facility and the Execution Control program for running simulations. The debugging facility is essential for developing the operational programs. The Node Processor has been designed to facilitate the operation of a debugger. The instruction trace enable bit in the Processor Status Word and the hardware memory address breakpoint comparator provide the necessary "hooks" for a debugger.

The Execution Control Program is the software utility for controlling simulation runs. It must be capable of handling the network in realtime. A simplified communications protocol and hardware design are essential to keep the complexity of the program to a minimum. Polled operation was selected over interrupt or asynchronous operation for the communications because of the need for a simple hardware and software design.

Node Processor Operational Program

The Node Processor Operational Programs are needed to execute the substructure simulations within each processor. Full software specifications for the operational programs are not possible until a sample problem is selected and the Model Specification Language is defined. The need for the operational programs is clear. Different sets of programs are needed for the various models the Digital System is to handle.

Node Processor Instruction Set

The Node Processor Instruction Set was designed as a comprehensive instruction set to accommodate the requirements of the simulation problems. The identified needs of the instruction set included:

1. General purpose instructions for flexibility in programming the Node Processor.
2. Matrix and vector operations for solving the state equations.
3. High speed floating point operations on double precision operands.

4. Comprehensive addressing modes for handling the data structures required by the simulation programs.

5. Microcoded parallel operation during lengthy calculations and I/O with nearest neighbors.

The instruction set designed has incorporated the needs identified above. It includes a full complement of instructions typically found on the latest generation of minicomputers. The many addressing modes and opcodes specified provide over 10,000 distinct operations. The vast capability did not slow the expected execution speeds of instructions since the hardware incorporates the latest techniques in pipelined architecture. In addition, the proper selection of instruction decoding and execution techniques enabled the extensive use of microcode subroutining, thereby keeping the size of the operational microcode within reason.

The higher level instructions specified for the Node Processor all take advantage of parallel operations where possible. Software pipelining [2] techniques were used in instructions which do matrix multiplication, gaussian elimination, sine/cosine calculation and Runge-Kutta-Merson integration. The use of software pipelining was possible because of the autonomous floating point multiplier and adder units.

In addition to using software pipelining wherever applicable, the floating point units were optimized for the operations expected during simulations. Floating point calculations are done on mantissas with 7 extra bits of precision so as to reduce round off error during intermediate calculations. The exponent range is the same in the floating point units as main memory. To speed up calculations on large matrices which contain many zero valued elements the floating point units detect zero operands on input and return their result immediately. When either operand is zero, the floating point

result is returned up to 70% faster.

SOFTWARE ASSESSMENT

A very powerful custom-made processor instruction set was designed and flowcharted. These instructions range from basic moves and integer operations to complex floating point operations with matrices. Since the node processor is microprogrammable, the suggested instruction set is not cast in concrete. The instructions were designed with simulation in mind and will make the solution of simulation models as efficient as possible.

Software routines for all levels of the system, however, are presently only in the conceptual stage. Offline model development software, real-time execution control software, and node processor operation programs remain to be designed. Each section of the software is in itself a substantial task.

The offline model development software will be made up of a compiler for the model specification language, an assembler and a linker for the node processor instruction set. It is not likely that an existing language will satisfy the need for the model specification language, so it must also be designed. The compiler for this language will translate the user model into node processor assembly language while segmenting the problem equally among the node processors. The assembler and linker would be fairly standard and similar to many available for minicomputers.

The realtime model execution software will consist of an execution control program which controls loading of the node processors and the synchronization of the processor array during run time. It also will collect data from the processors when appropriate. A node processor program debugger will be a part of this software to aid in the development of operational programs.

The node processor operational programs will be the true simulation programs. These programs will be dependent on the simulation model.

The node processor microcode, while flowcharted, remains to be written. These routines, sometimes called firmware, will implement the node processor instruction set.

Above and beyond all of the simulation programs, it will be necessary to have diagnostic software to insure the proper operation of the control computer with the node processor array. Lower level programs should be capable of identifying board level faults within a node processor.

The above software assemblage will provide a functionally complete and convenient package for structural dynamics simulation.

NODE PROCESSOR HARDWARE

The Node Processor hardware consists of seven major blocks of hardware:

1. The microprogram controller has the function of decoding program instructions and controlling the proper operation of the remaining six sections of hardware.

2. The registered arithmetic logic units (RALU's) perform the integer arithmetic and logic functions of the node processor. The RALU's are the BIT-SLICES.

3. The dynamic memory is the main storage area for local program and data storage within the node processor.

4. The node processor communications hardware provides the I/O ports necessary for the six nearest neighbor communications and communications with the central minicomputer.

5. The floating point bus interface and scratch pad is used between the CPU of the node processor and the floating point units to buffer, hold intermediate floating point values, and expand/truncate floating point values.

6. The floating point multiplier is used to perform all floating point multiplies within the node processor.

7. The floating point adder/subtractor/divider performs all floating point addition, subtraction, and division within a node processor.

The relationship of the above hardware is shown in Figure 6.

Each node processor has its CPU implemented with microprogrammed bit-slice hardware. Bit-slice hardware is currently available in ECL or TTL technology. TTL technology was chosen because of difficulties in designing with ECL. ECL consumes a great deal of power, the variety of circuits available is limited, and second sourcing of parts is a problem.

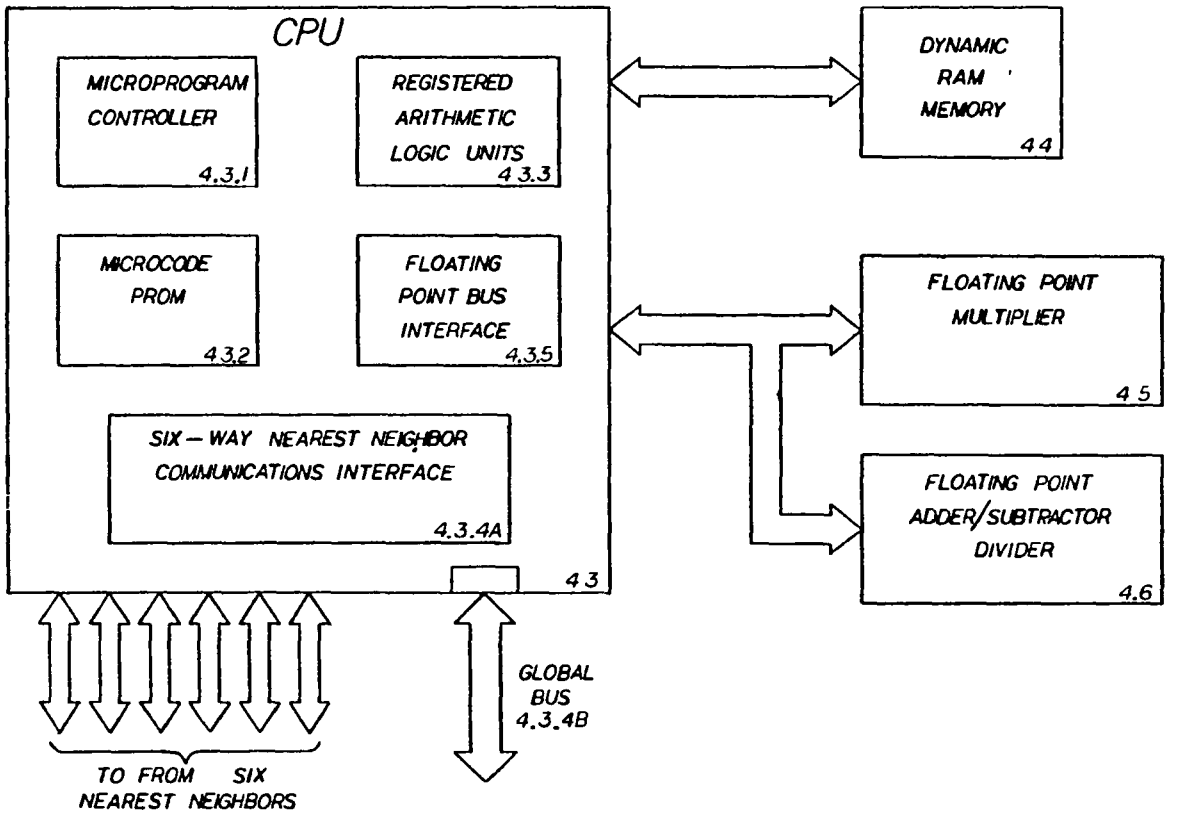


Figure 6 NASA/Lewis Simplified Node Processor Block Diagram

In TTL technology, Advanced Micro Devices (AMD) is the leader in bit-slice hardware. Other manufacturers second source many of AMD's products. Standard TTL, lower power Schottky, Schottky and new advanced Schottky are all compatible with the bit slice hardware. The most flexible, low power power, high speed design is possible with TTL technology.

MICROPROGRAM CONTROLLER

The microprogram controller is the section of the node processor CPU which selects a coherent sequence of microinstructions used to execute the various instructions required by the processor. Each elemental task performed by the processor is called a microinstruction. A single machine instruction will take one or more microinstructions to execute. These microinstructions are stored in a permanent memory called microcode PROM. A sequence of microinstructions initiated by a machine instruction is called a microroutine. Because there is a great deal of functional overlap, many machine instructions will execute microroutines that share portions of the microcode.

The node processor microprogram controller consists of the following hardware: the Instruction Register, the Instruction Mapping Prom, the Address Mode Mapping Prom, an Address MUX, the Microprogram Sequencer, the Condition Code Select MUX, the Microprogram Memory, and the Pipeline Register. Also shown on the block diagram of Figure 7 is the Clock Generator.

Instruction Register

The Instruction Register is a 64-bit edge triggered register which holds the next machine instruction to be executed. It takes two microcyles to fetch the instruction from the dynamic memory and load it in the IR. Each microcycle may only fetch 32 bits. Since the instruction is so wide, it contains 7 different fields capable of implementing a very complete instruction set.

OP CODE 8 BITS	MODE SELECT #1 4 BITS	REGISTER SELECT #1 4 BITS	MODE SELECT #2 4 BITS	REGISTER SELECT #2 4 BITS	INDEX #1 20 BITS	INDEX #2 20 BITS

INSTRUCTION REGISTER

The 8-bit op code allows for up to 256 different op codes. Two registers and their addressing modes may be selected. Two separate index values may be specified. Of course, not all instructions will use all fields and the format may vary slightly among instructions.

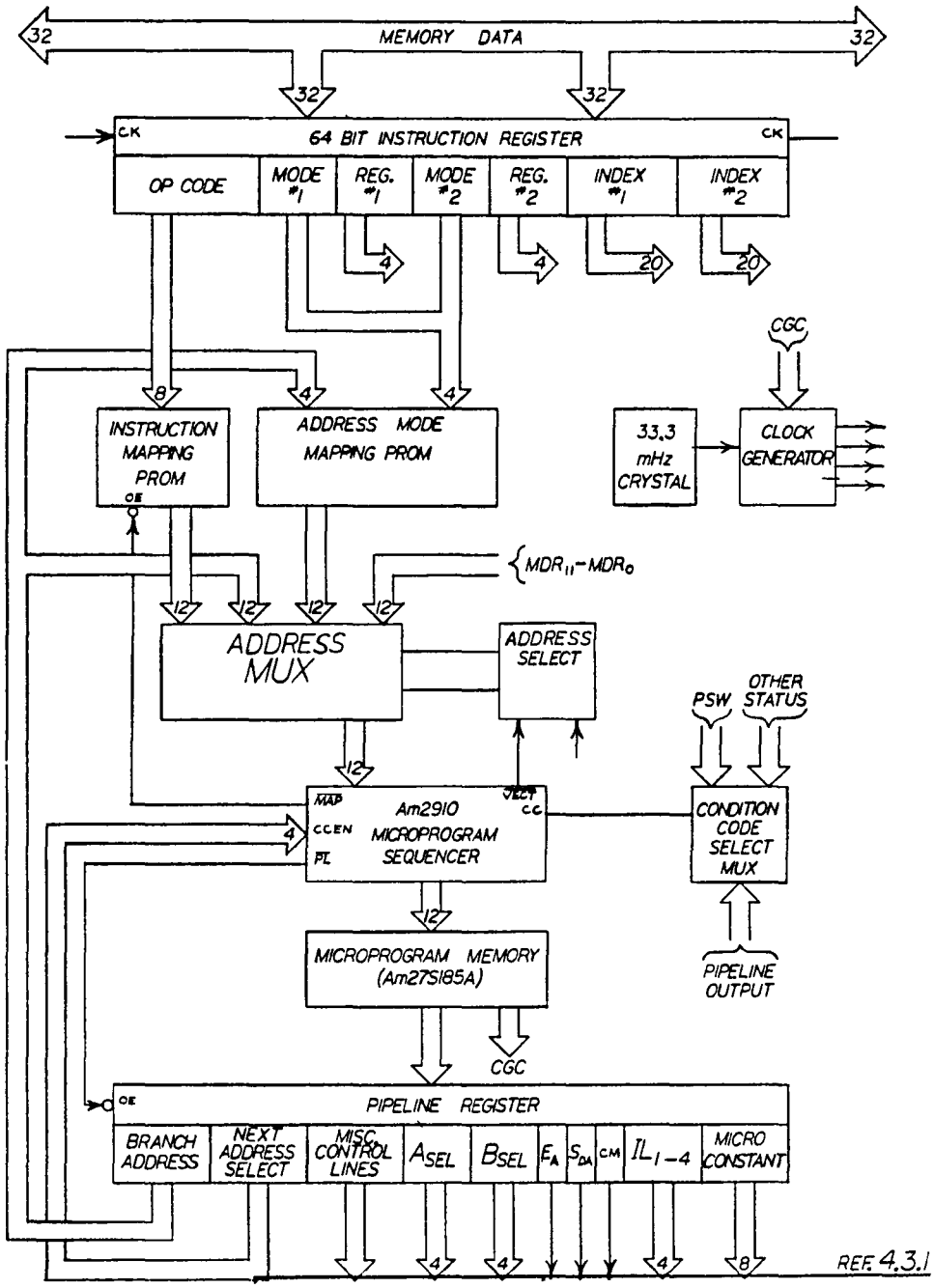


Figure 7 NASA/Lewis SDS Node Processor CPU
Microprogram Controller

Instruction Mapping PROM

The Instruction Mapping PROM is a code converter which converts the eight bit Op Code into a 12-bit microaddress. It is a 256 x 12 bit wide area of PROM. The 12-bit microaddress is typically the start of the micro-routine for the given microinstruction.

Address Mode Mapping PROM

The Address Mode Mapping PROM is a code converter which takes 4 bits from the Branch Address field of the Pipeline Register and 4 bits from one of the two address mode select fields of the instruction register and converts it into a 12 bit microaddress. Only several of the possible 256 different addressing modes are actually implemented.

Address MUX

The Address MUX selects one of four different branch addresses to the Am 2910 microprogram Controller. The four choices are the Instruction Mapping PROM, the Branch Address field of the Pipeline Register, the Address Mode Mapping PROM, or the least significant 12 bits of the Memory Data Register (MDR_{11}-MDR_0).

Microprogram Controller

The Am 2910 Microprogram Controller [3] is an address sequencer intended for controlling the sequence of execution of microinstructions stored in microprogram memory. Besides the capability of sequential access, it provides conditional branching to any microinstruction within its 4096-microword range. A last in, first out stack provides microsubroutine return linkage and looping capability allowing five levels of nesting subroutines.

Condition Code Select MUX

The Condition Code Select MUX selects the branch condition to the CC input of the Am 2910 Microprogram Controller. The conditions are:

1. Floating Point Multiplier
    a. ZERO
    b. CARRY
    c. NEGATIVE
    d. OVERFLOW
2. Floating Point Adder
    a. ZERO
    b. CARRY
    c. NEGATIVE
    d. OVERFLOW
3. Integer CPU
    a. ZERO
    b. CARRY
    c. NEGATIVE
    d. OVERFLOW
4. (Floating Point Trap Enable) AND (Floating Point Multiplier Overflow)
5. (Floating Point Trap Enable) AND (Floating Point Adder Overflow OR Divide by Zero)
6. Trace bit set in Processor Status Word
7. Memory Error
    a. One Bit Error
    b. Two Bit Error
8. Floating Point Multiplier DONE
9. Floating Point Adder/Subtractor/Divider DONE
10. ALL READY on Global Bus
11. Data Received on input latch
12. Memory Write Fault (Memory Address Register $\leq$ Address Fence) on write operation
13. MEMORY TRAP: (One Bit Error) OR (Two Bit Error) OR (Write AND Memory Address Register $\leq$ Address Fence)

Microprogram Memory

The Microprogram Memory contains all of the microroutines which form the instruction set of the node processor. Since the microcode word for the node processor has not been completely defined, nor has the microcode been written, the width and depth of the PROM area is not specified. Due to hardware development constraints the microcode depth is limited to 4K words and it is preferable to keep the width at 64 bits or fewer.

Two possible choices for the PROM chips are the Am 275185A (2K x 8) or the Intel 3632 (4K x 8).

Pipeline Register

The Pipeline Register allows an overlap of the fetch of the next microinstruction while the current microinstruction is being executed. The next instruction is being decoded while the microinstruction latched in the pipeline register is being executed. The position of the pipeline register immediately after the microprogram PROM causes this arrangement to be called the instruction - data based architecture.

Clock Generator

The Clock Generator is the Am 2925 System Clock Generator and Driver. This integrated circuit is programmed by the microcode in order to vary the microcycle length. It also contains the HALT/RUN and SINGLE STEP logic used for system debugging. Up to 8 different cycle lengths of the four phase output may be generated.

REGISTERED ARITHMETIC AND LOGIC UNITS (RALU's)

The RALU's are the hardware where the integer arithmetic and the logic operations of the node processor are performed. The RALU's have associated with them Register Select Input Multiplexers, Direct Input Select Multiplexers, Shift Control Logic, Memory Address Register, Memory Data Register, Memory Data Bidirectional Buffer, Processor Status Word Multiplexer, and Processor Status Word Register. See Figure 8.

The above hardware is responsible for the generation of the next macro-address and together with the Microprogram Controller, forms the node processor CPU.

RALU's

The RALU's are the Am 2903 four bit bipolar microprocessor slices. There are eight of these bit slices for a 32-bit CPU. The RALU's perform all of the integer arithmetic such as memory address and all of the logical functions needed by the CPU. The RALU's contain 16 internal dual port registers, various internal latches, and shifters.

REGISTER SELECT INPUT MULTIPLEXERS

The Register A Select Input Multiplexer selects one of five different groups of 4 bits to be the A Register address. The five bit fields are the REG #1 instruction field, the least significant 4 bits of INDEX #1 or INDEX #2 instruction field, or the A SEL field of the Pipeline Register, or the MODE 2 field.

The Register B Select Input Multiplexer selects one of four different groups of 4 bits to be the B Register address. The four bit fields are the REG #1 or REG #2 instructions, the four least significant bits of the Memory Data Register, or the B SEL Pipeline Register field.

DIRECT INPUT SELECT MULTIPLEXERS

The Direct A Input Select Multiplexer selects one of four different sets of

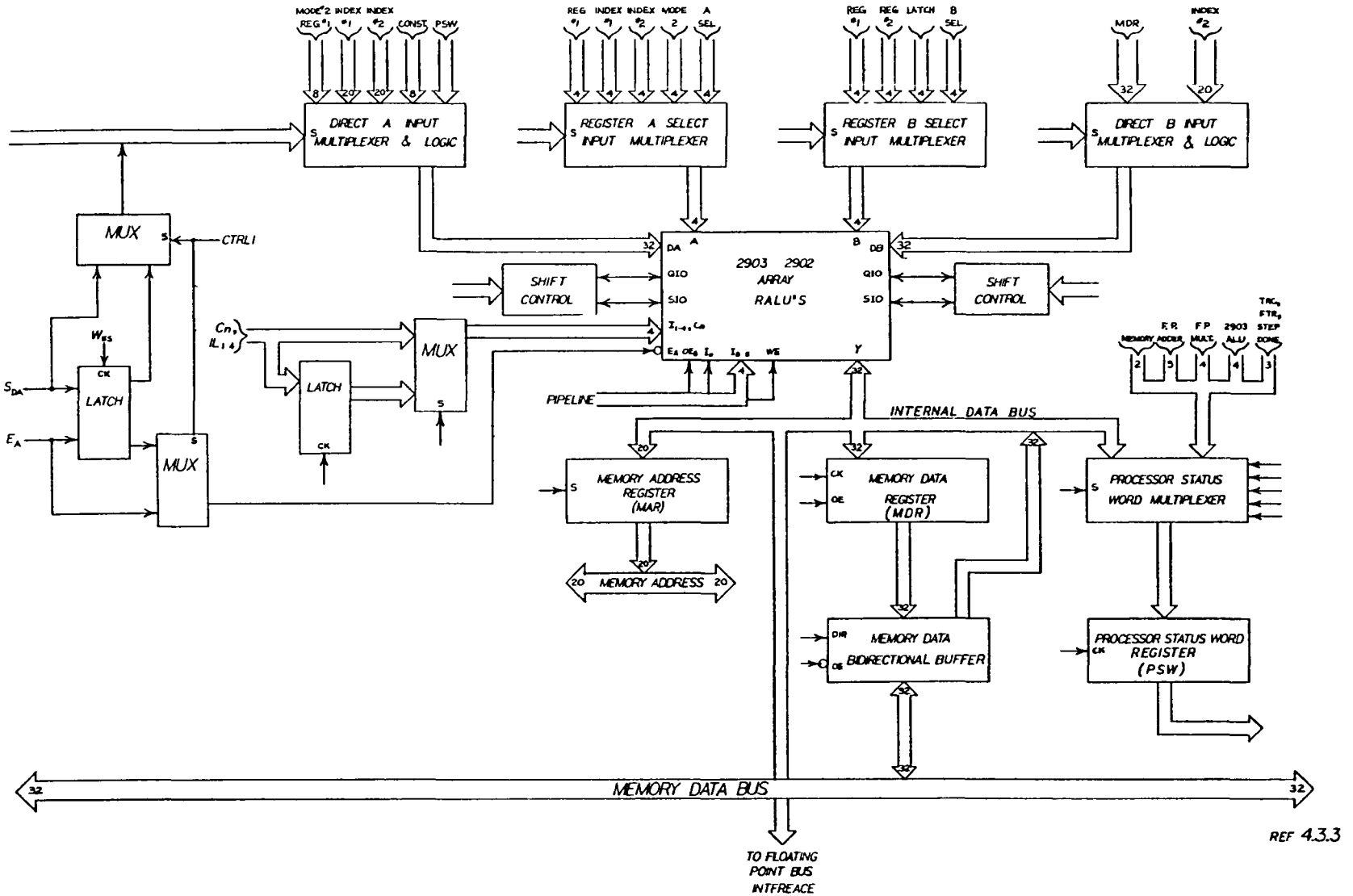


Figure 8 NASA/Lewis SDS Node Processor CPU Registered ALU's Block Diagram

inputs to the DA 2903 inputs. These fields are the 8 bit field formed by the Index #1 or Index #2 Instruction Register fields, an 8 bit latched constant, or the Processor Status Word.

Shift Control Logic

The Shift Control Logic determines the type of shift done by the Am 2903. A one or a zero may be selected as the shift input, or a rotate may be selected.

Memory Address Register (MAR)

The MAR is a 20 bit wide positive edge triggered register used to hold the current dynamic memory address. This register also buffers the address onto the Address lines.

MEMORY DATA REGISTER (MDR)

The MDR is a 32 bit wide positive edge triggered register used to hold values to be placed on the MEMORY DATA BUS.

MEMORY DATA BIDIRECTIONAL BUFFER

This buffer buffers the 32 bit wide output of the MDR to the MEMORY DATA BUS and it buffers the data from the MEMORY DATA BUS to the INTERNAL DATA BUS.

PROCESSOR STATUS WORD MULTIPLEXER

The PSW multiplexer selects between the Internal Data Bus and the various Processor Status bits as inputs to the PSW register.

PROCESSOR STATUS WORD REGISTER (PSW)

The PSW holds information concerning the current status of the node processor. The PSW bits are defined as follows:

PROCESSOR STATUS WORD

Bit	Mnemonic	Contents
0	CR	Integer carry
1	OV	Integer overflow
2	NE	Integer result negative
3	ZE	Integer result zero
4	FUN1	FP multiply underflow
5	FOV1	FP multiply overflow
6	FNE1	FP multiply negative
7	FNE1	FP multiply result zero
8	FUN2	FP add/subtract/divide underflow
9	FOV2	FP add/subtract/divide overflow
10	FNE2	FP add/subtract/divide result negative
11	FZE2	FP add/subtract/divide result zero
12	FDZ	FP divide by zero
13	FTR	Trap enable on FDZ, FOV1, or FOV2 via location 10
14	TRC	Trace enable. Trap after each instruction.
15	PAR	One bit error from memory access
16	RER	Two bit error from memory access
17	RDY	Write only. Signals to host this processor is done with current step.
18	BPT	Enable trap on setting of FEN
19	FEN	Memory write address equals MACR
20	FLT	Memory write address less than MACR

DYNAMIC MEMORY

Each dynamic memory board contains 256K x 32 bits of memory (TMS 4164), a dynamic memory controller, memory and refresh timing, and a section for error detection and correction as shown in Figure 9.

A maximum of four (4) boards may be addressed by each node processor. A node with four memory boards would have 1 megaword (32-bit words). Except for CPU registers and floating point registers, all of the program and data values used by the node processor are stored in the dynamic RAM.

Each board contains 156 64K dynamic RAMS. The memory array has 128 integrated circuits while the error detection and correction section uses the remaining 28.

The Am2964B Dynamic Memory Controller is used to provide all address handling, as well as $\overline{RAS}$ and $\overline{CAS}$ decoding and control. The device has 18 input latches for capturing an 18-bit address for memory control. The two highest order addresses are used to select one of four 64K x 32 bit blocks of RAM. The Am2964B also contains an 8-bit refresh counter used to provide the necessary 256 line refresh mode. The $\overline{CAS}$ output is inhibited during refresh.

Normal operation of the Dynamic Memory Controller is to provide the address, close the address latches and start off a normal memory cycle. This is accomplished by bringing the $\overline{RASI}$ input LOW which will cause one of the $\overline{RAS}$ outputs to go low. After the required memory timing, the MSEL input is used to switch the multiplexer to the $\overline{CAS}$ latch. Then the $\overline{CASI}$ input will be driven LOW and execute the $\overline{CAS}$ part of the memory cycle.

The refresh cycle is executed by driving the $\overline{RFSH}$ signal low which causes all four $\overline{RAS}$ outputs to go low. This will simultaneously refresh all four banks of memory controlled by the Dynamic Memory Controller. When either the $\overline{RFSH}$ or $\overline{RASI}$ input is brought high, the refresh counter is advanced so it will be ready for the next cycle.

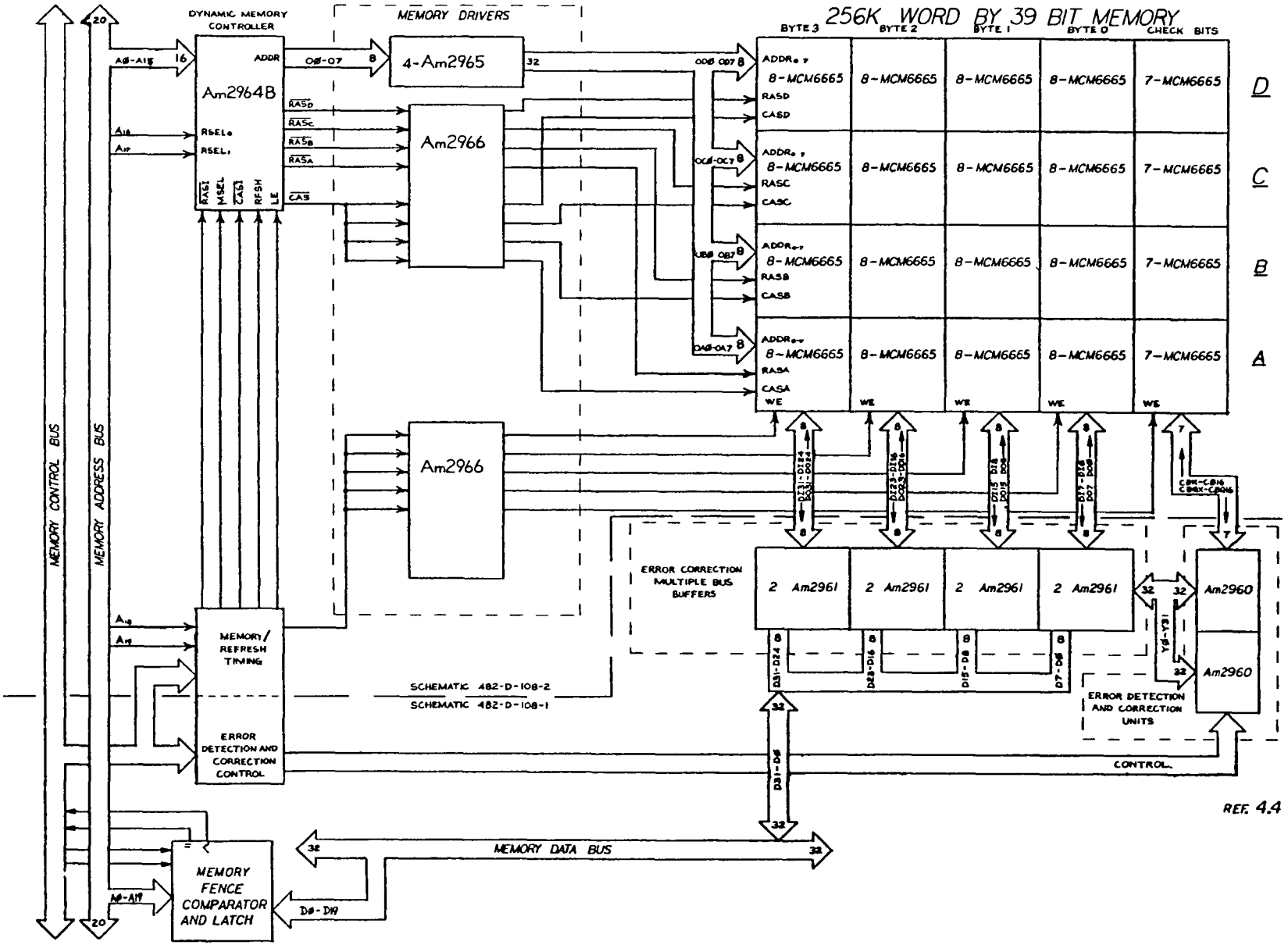


Figure 9 NASA/Lewis Node Processor CPU Dynamic Memory 256K WD x 39 Bit

Data interface among the dynamic memories, the Am2960 EDC circuit, and the node processor data bus is accomplished by means of the Am2961 bus buffers. Each 2961 contains two internal latches, a multiplexer, and a RAM driver output buffer. Each 2961 is 4 bits wide so 8 are used in this 32-bit system.

The bus input latch of the 2961 is used for data storage during a memory WRITE. The bus output latch in the 2961 is used predominantly for storing the output data if the processor is in single step mode. In the single step mode it is necessary to hold the output data on the system data bus but the memory must be free to be refreshed.

A pair of Am2960 Error Detection and Correction units (EDC's) contain all the necessary logic to generate check bits on a 32-bit data field according to a modified Hamming code and to correct the data when the check bits are supplied. Operating on the data read from memory, the EDC's can correct all single bit errors and will detect all double and some triple bit errors. For 32-bit words, 7 check bits are used.

Some additional circuitry is required to provide proper memory access sequencing and timing for memory refresh. Since the 16 bits of address must be multiplexed into the dynamic RAMs 8 address lines, the memory timing is necessary. It is necessary to allow the memory to be refreshed, with an eight bit address, when it is not being accessed. When the CPU is running, refresh is automatic and transparent within the microcode sequences. When the CPU is halted, such as during single step mode, a special refresh counter periodically refreshes the RAM.

Aside from the two refresh modes described above, the memory normally has three operating modes. In the write mode, a 32-bit value is loaded into the data input latch. The 7 check bits are generated by the EDCs which correspond to the 32-bit value. At the end of the write cycle, the data and the check bits are written into the proper RAM location.

In the detect mode the EDCs examine the contents of the Data Input latch (from the RAM) against the Check Bit Input Latch, and will detect all

single bit errors, all doublt-bit errors, and some triple bit errors. If one or more errors are detected, the $\overline{ERR}$ status line to the CPU is pulled low. If two or more errors are detected, $\overline{MERR}$ is pulled low. Both $\overline{ERR}$ and $\overline{MERR}$ are open collector signals that remain high if there are no errors. In the Detect mode, the contents of the Data Input latch are driven directly to the Data Output Latch without correction.

In the Correct mode, the EDCs function the same as in the Detect mode except that the correction network is allowed to correct (complement) any single bit error of the Data Input Latch before putting it into the inputs of the Data Output Latch. If multiple errors are detected, the output of the correction network is unspecified, and both the $\overline{ERR}$ and $\overline{MERR}$ lines are pulled low. If the single-bit error is a check bit, there is no authomatic correction; if desired, this would be done by placing the EDCs in generate mode to produce the correct check bit sequence for the data in the Data Input Latch.

An option on the memory board is the Memory Fence Comparator. These integrated chips should only be installed on one memory board per node. If present, a special instruction called the Write Fence Instruction will load an immediate 20-bit value into the Fence register. Whenever memory is written, the address is compared to the 20-bit Fence value. If address equals fence the status line EQAD is brought high. If address is less than fence the status line FENCE is brought high. The EQAD status line finds use as a means of generating a hardware breakpoint. The FENCE status line finds its use in detecting illegal memory writes.

The Memory Fence feature is used to insure that an area of dynamic memory has not been overwritten by mistaken. For example, a function look up table may have been loaded into the lower memory area and the Memory Fence is set at the top of this table. If a programming error or a hardware error forced a write to this area of memory, the FENCE line would be brought high. This FENCE signal would be detected and warn the system operator

that the resulting computations in the node processor may have been corrupted. The EQAD line is brought high when the Memory Fence value equals the memory address. This feature is used as a hardware break point during program debugging.

NODE PROCESSOR COMMUNICATIONS

The Node Processor Communications hardware is the most difficult part of the system architecture to define. (Unless otherwise noted, Node Processor Communications refers to communications between a node and its six nearest neighbors.) There are three schemes for this communication: shared memory, FIFO Buffers and six way communications controller. Each is discussed in the following paragraphs.

Shared Memory

A system with shared memory between nearest neighbors would enable a node to directly access a section of its neighbor's memory. This is a fast, perhaps the fastest possible, method of data transfer between processors. There are several important disadvantages. The memory for such a system is more complex; it would require dual port memories that are expensive. The memory width is 32 bits plus address and control lines. In dealing with six nearest neighbors, more than 200 connections would be needed on each processor to implement shared memory. A large number of interconnects are not desired because of low reliability.

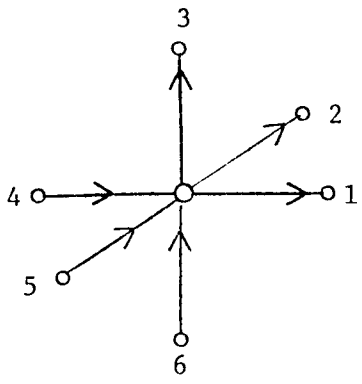
FIFO Buffers

A second method uses narrower data paths (16 bits per transfer) with a FIFO buffered input and output. Data are transferred between boards in two separate events or passes. Pass one consists of loading three output FIFOs and unloading three input FIFOs. Pass two changes the direction of the data transfer. Each pass sends data to three nearest neighbors and receives data from the remaining three neighbors.

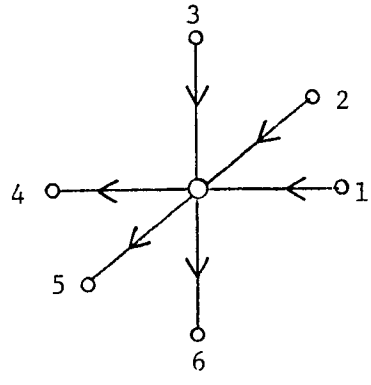
For example, during pass one, data are passed from processor N to neighbors 1, 2, and 3. Data are received from neighbors 4, 5 and 6. During pass two, data are received from neighbors 1, 2, and 3. Data are sent to neighbors 4, 5, and 6. See Figures (Pass 1 and Pass 2) on page 74.

The maximum number of variables sent along the path from one processor to its neighbor is 100. Each variable is a 64 bit number. The path width chosen for data transfers is 16 bits. A FIFO depth of 400 is needed to hold the variables during any single path. The block diagram for the FIFO scheme is shown in Figure 10.

The above two methods of processor communications are quite costly in terms of the amount of hardware involved. Upon reference to Sample Problem 2 (Determination of Processor Computational Capabilities) implementation of either scheme seems unjustified. If 8×10^6 floating point computations were performed for each time step at a cost of 2 μsec per operation (being very optimistic about the speed) the compute time would be 16 sec. The time it would take a node to output 100 variables, 16 bits at a time, 500 nsec. per transfer, (being conservatively slow), then input 100 variables in the same manner would be 400 μsec Even if data transfer occurred that slowly, nearest processor communications would only represent .0025% of a single time step. This leads to the conclusion that communications may be performed adequately with less hardware by a simpler set of six input/output ports.



Pass 1



Pass 2

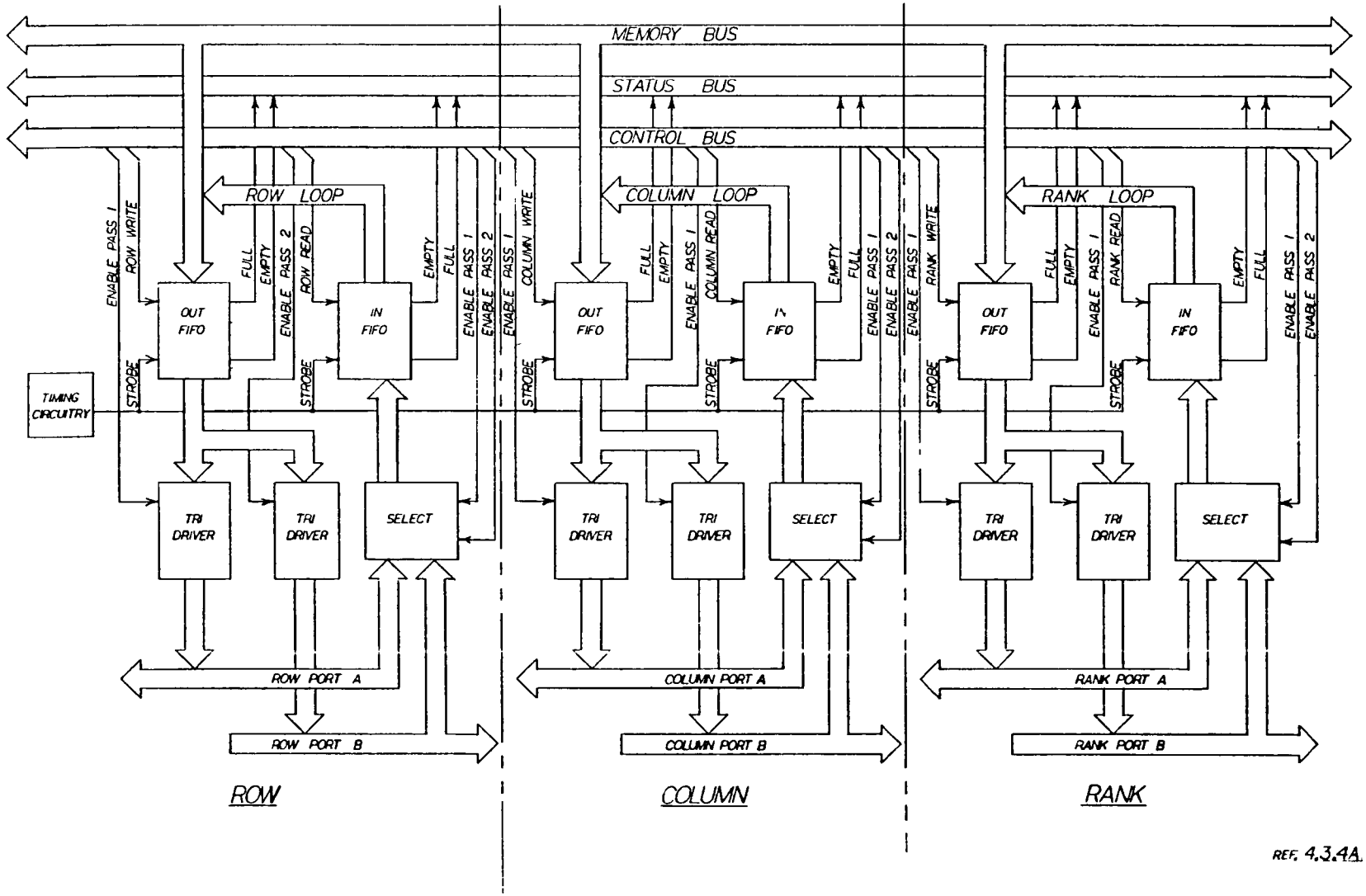


Figure 10 NASA/Lewis SDS Node Processor CPU Block Diagram Six-Way Communications Interface

Six Way Communications Controller

The six way communications controller is a set of six I/O ports under control of the node processor CPU. Except for the address, all six ports are identical. Each port is hardwired to the appropriate port on the six nearest neighbors of the node. See Figure 11.

Each port consists of two 16-bit output latches with a common clock and separate output enables. There are two 16-bit input registers with separate clocks and a common output enable.

Output of data to the nearest neighbor is accomplished by fetching a 32-bit value and latching it in the output latch. The high order half word (16 bits) is sent when the CPU receives the $\overline{\text{EMPTY}}$ status signal from its nearest neighbor. The low order half word is sent in the next microcycle. To send a 64-bit value this procedure must be done twice. Figure 12 is a flow chart of the output loop of the six way communications controller. Figure 13 depicts the typical output operation of the controller.

Input of data from the nearest neighbor can occur when the $\overline{\text{EMPTY}}$ status signal is sent to the nearest neighbor. First the high order half word is clocked into the high order input register. In the next microcycle, the low order half word is latched into the low order input register and the $\overline{\text{EMPTY}}$ signal is set (=1). Once $\overline{\text{EMPTY}}$ = 1 the CPU may take the data and put it in its intended destination. Figure 14 is a flow chart of the input loop of the six way communications controller. Figure 15 depicts the typical input operation of the controller.

There are three signals used for handshaking between nodes. When $\overline{\text{EMPTY}}$ is active low, the input port is ready to be loaded from the nearest neighbor. The other two handshake lines clock the data into the input register of the nearest neighbor. The two clocks CKH and CKL clock the upper and lower halves of a typical transfer into the input register of the nearest neighbor.

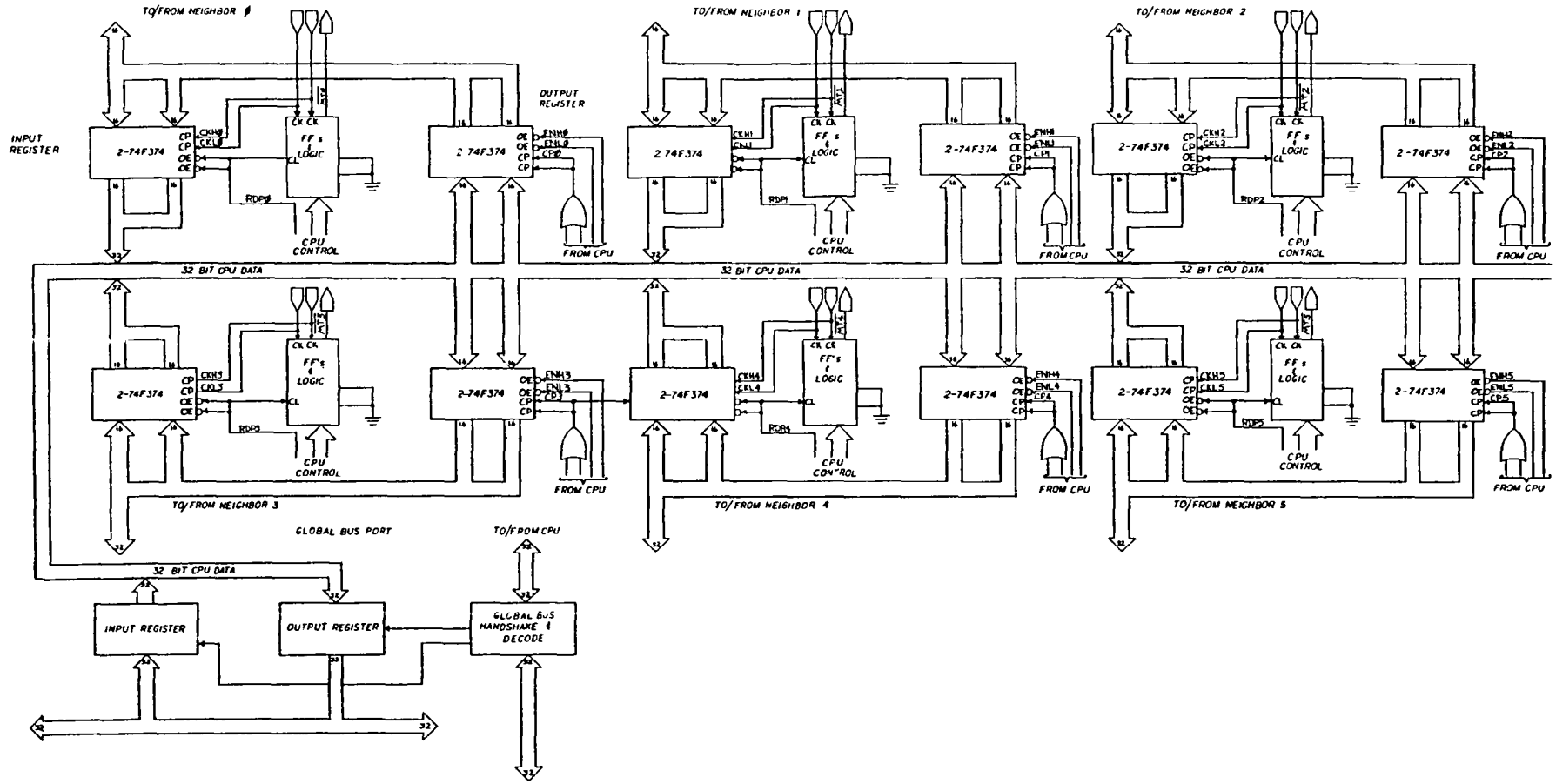


Figure 11 NASA/Lewis Six-Way Communication Interface Block Diagram

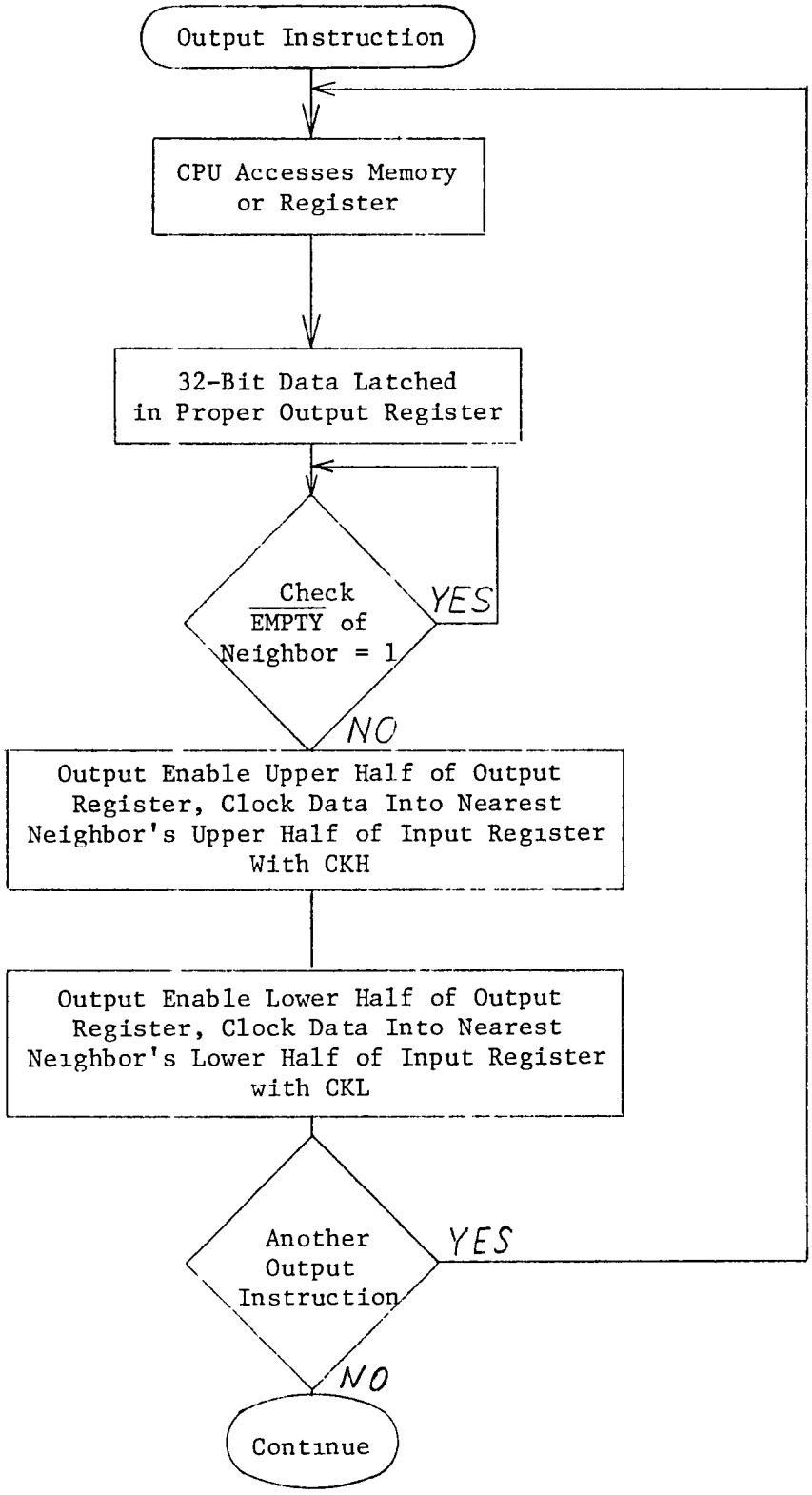


Figure 12 Six Way Communications Controller, Output Loop

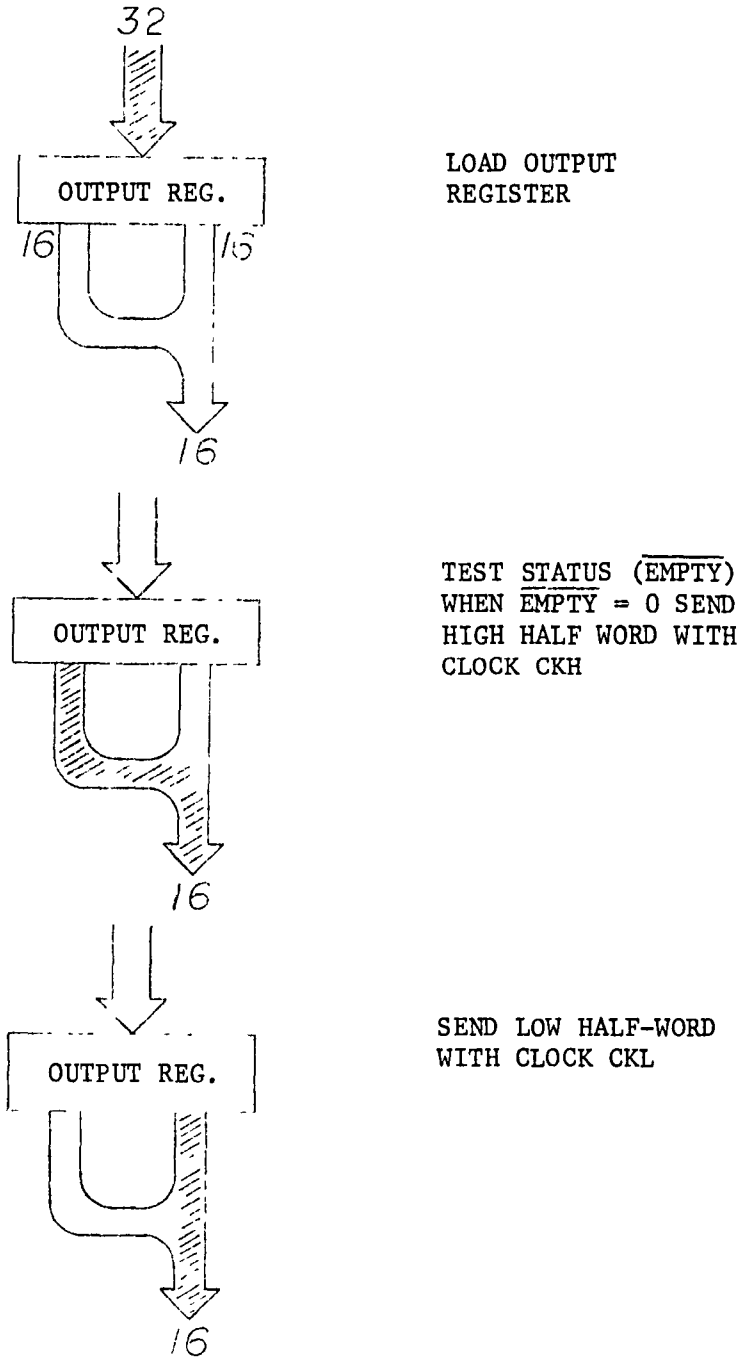


Figure 13 Output Operation of the Six Way Communications Controller

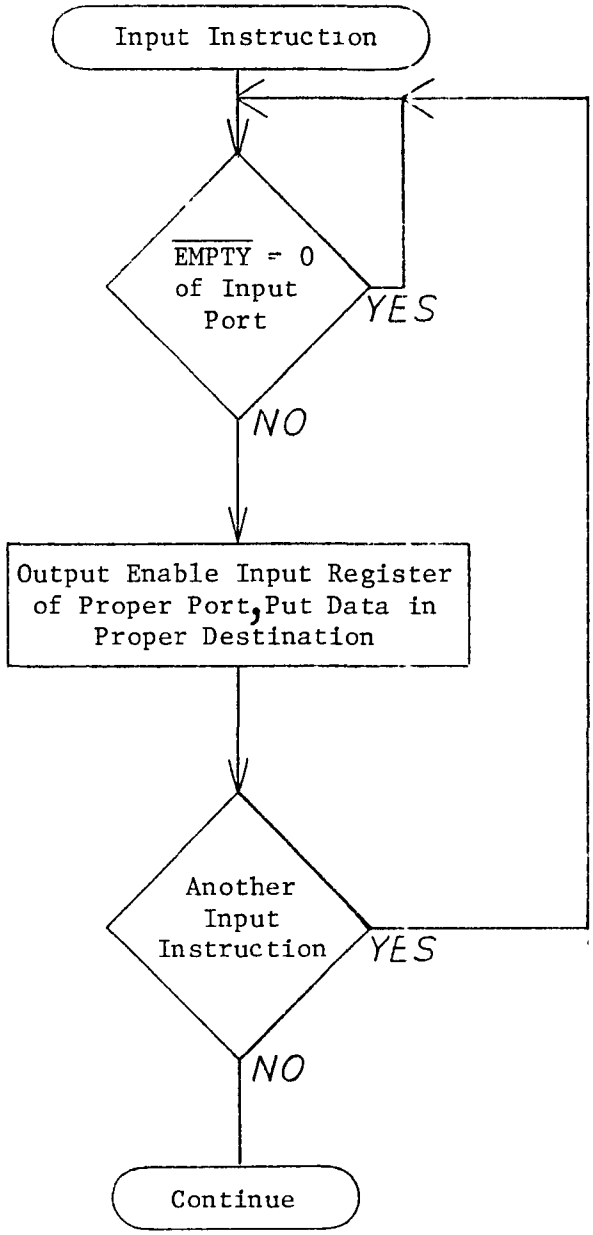
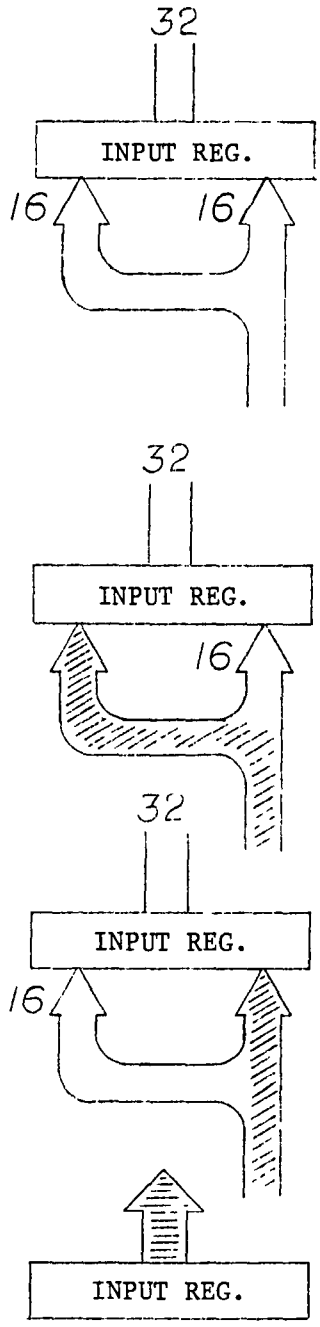


Figure 14 Six Way Communications Controller Flowchart, Input Loop



ASSERT $\overline{\text{EMPTY}}$ = 0 WHEN READY FOR INPUT

THE HIGH ORDER HALF WORD IS SENT AND LATCHED BY THE ADJACENT NODE

THE LOW ORDER HALF WORD IS SENT AND LATCHED BY THE ADJACENT NODE. $\overline{\text{EMPTY}}$ = 1 (Not empty)

$\overline{\text{EMPTY}}$ = 1 IS DETECTED BY THE CPU AND THE DATA IS READ FROM THE INPUT LATCH $\overline{\text{EMPTY}}$ IS SET TO 0.

Figure 15 Input Operation of the Six Way Communications Controller

Global Bus Communications Port

Each node processor has a single 32-bit bidirectional port known as the global bus port. This port is used for communications between any node and the control computer.

At the node processor, the port consists of a 32-bit input latch, a 32-bit output latch, and a status flip flop.

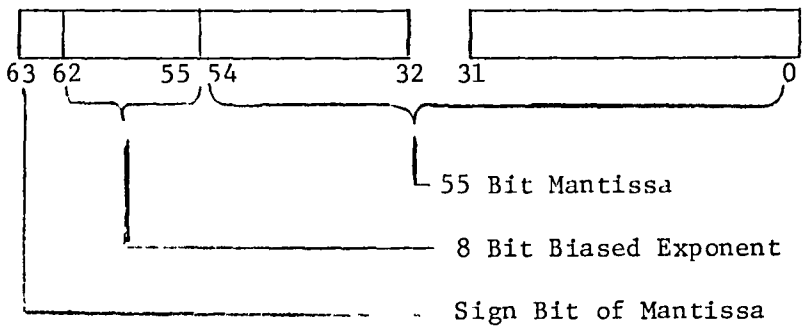
The type of handshake to be used has not been decided at this time.

FLOATING POINT BUS INTERFACE (FPBI) AND SCRATCH PAD

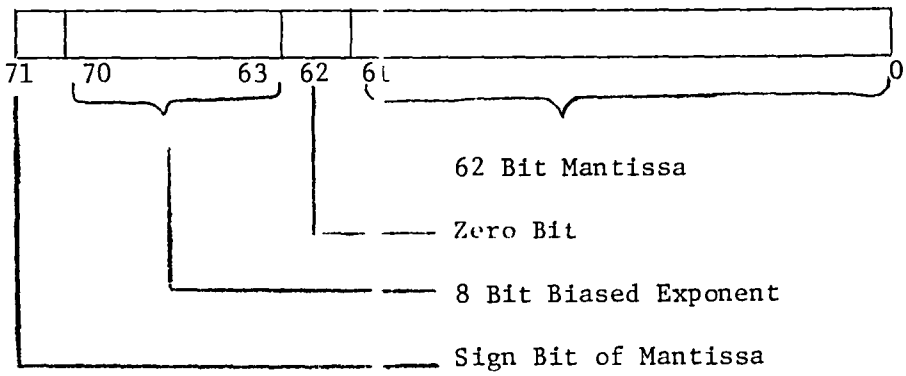
The FPBI and Scratch Pad is designed to buffer, format and store data between the node processor CPU and Floating Point units (i.e., Floating Point Multiplier and Floating Point Adder/Subtractor/Divider).

Typical CPU floating point values are 64-bits wide and take up two 32-bit wide words of CPU memory. On the other hand, the Floating Point units both handle 72-bit wide floating point values. Hence the need for a CPU to Floating Point unit interface. The two Floating Point word formats are shown below:

64 BIT CPU Floating Point Value

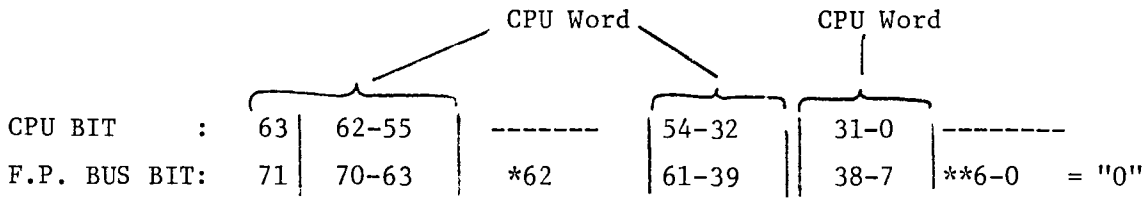


72 BIT Floating Point Bus Value



CPU to FPBI Transfers

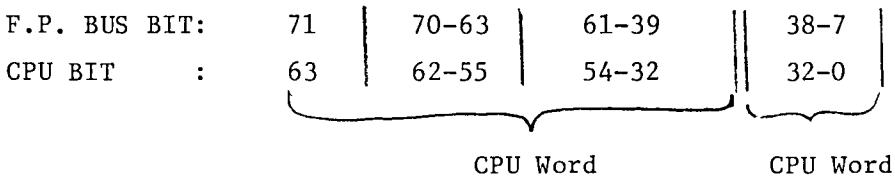
As shown above the FPBI must accept two 32-bit values from the CPU and expand this value to 72 bits. The mapping of this expansion is shown below:



*F.P. Bus Bit 62=0 if bits 62-55 are all zero; else 62=1.
**F.P. Bus Bit 6-0 are all set to "0".

FPBI to CPU Transfers

Bit 62 and bit 6-0 of the FBPI word are truncated on a transfer from the FPBI TO THE CPU:



SCRATCH PAD Area

The SCRATCH PAD area of the FBPI is a 256 x 72 bit wide area of fast static RAM. The SCRATCH PAD serves several purposes: 1) It allows quick access of a commonly used operand (i.e., parallel access to all 72-bits versus two accesses to the slower CPU Dynamic RAM); 2) Greater precision is maintained in the 72-bit intermediate value; 3) In matrix operations, a whole row of a matrix may be stored for convenient access.

Operating Details of FPBI

Write to SCRATCH PAD from CPU

Step 1: An 8-bit address is sent to the F.P. address transparent latch while the least significant 32-bits of the F.P. value is latched in an input latch.

Step 2: The higher order 32-bits of the F.P. value are sent along with the write command which causes the proper 72-bit value to be written in the scratch pad at the appropriate address.

Read from SCRATCH PAD to CPU

Step 1: An 8-bit address is sent to the F.P. address transparent latch and the upper 32-bits are read in on the data bus while the lower 32-bits of the F.P. word are latched.

Step 2: The lower 32-bits are read in from the latch to complete the transfer.

Write from SCRATCH PAD to FLOATING POINT BUS

An 8-bit address is sent to the F.P. address transparent latch, the F.P. bus drivers are enabled, and a scratch pad Read is enabled. At the end of this step the appropriate register in a Floating Point Unit latches the F.P. value from the bus.

Read from FLOATING POINT BUS to SCRATCH PAD

An 8-bit address is sent to the F.P. address transparent latch, the F.P. bus receivers are enabled, a scratch pad write is enabled and the appropriate register in a Floating Point Unit is enabled on the bus. At the end of the cycle the result is latched into the scratch pad. F.P. status is also latched in the seven bit status latch.

REGISTER to REGISTER transfer

Step 1: An 8-bit (source) address is sent to the F.P. address transparent latch, a scratch pad Read is enabled and the output transfer latch stores the 72-bit floating point value.

Step 2: An 8-bit (destination) address is sent to the F.P. address transparent latch, the output transparent latch is output enabled, the F.P. bus receivers are enabled and a scratch pad write is enabled.

FPBI Hardware Description

The FPBI has five main parts: The floating point address latch, CPU memory data bus buffers and latches, an 8-bit comparitor, 256 x 72-bit scratch pad RAM, floating point bus buffers and latches. See Figure 16.

The floating point address latch holds the 8-bit address from the CPU of the scratch pad RAM. This latch is transparent which means it may be opened during one cycle and stored on subsequent cycles.

The CPU memory data bus buffers and latches are used to multiplex and truncate the F.P. data on a scratch pad read. Data is latched, buffered and expanded during a scratch pad write. Two cycles are required for a read or write to the memory bus since it is only 32-bits wide.

An 8-bit comparitor compares the exponent with zero and sets the zero detect bit on a write from the CPU to the scratch pad.

The 256 x 72-bit scratch pad is a fast read-write memory used to hold the expanded double word operands used in the floating point units.

The floating point bus output latch and input buffer isolates the F.P. bus from the F.P. UNITS and the F.P. scratch pad. There is a 72-bit buffer from the F.P. BUS to the scratch pad. There is a 72-bit transparent latch from

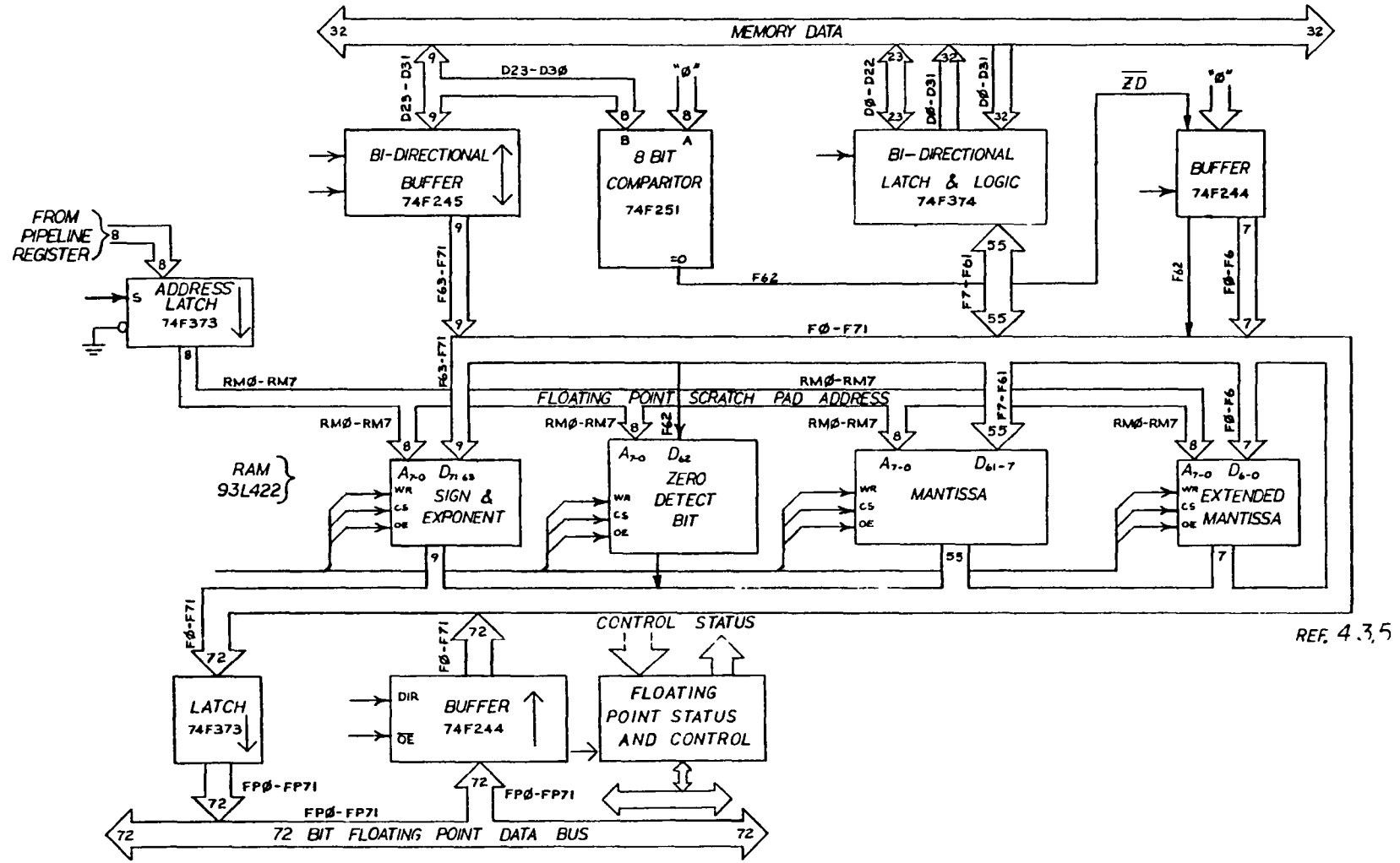


Figure 16 NASA/Lewis SDS Node Processor CPU Floating Point Bus Interface Block Diagram

the scratch pad to the F.P. bus.

The floating point control lines are derived from CPU microcode bits. The floating point status lines are latched in a seven-bit latch.

FLOATING POINT BUS CONTROL LINES

$\overline{\text{WRFP}}$ A low on this line is used to write data into a Floating Point Unit.

$\overline{\text{RDFP}}$ A low on this line is used to read data from a Floating Point Unit.

ADD/$\overline{\text{MULT}}$ A high on this line is used to access to the Floating Point Adder/Subtractor/Divider. A low on this line is used to access to the Floating Point Multiplier.

FPR2-FPRØ These lines determine the register of the Floating Point Unit accessed and in the case of the Adder/Subtractor/Divider, the function to be performed.

$\overline{\text{WRFP}}$	$\overline{\text{RDFP}}$	ADD/$\overline{\text{MULT}}$	FPR2	FPR1	FPRØ	
H	H	X	X	X	X	NO OP
L	H	H	L	L	L	Write X operand to Adder
L	H	H	L	L	H	Write Y operand to Adder - Function:Add
L	H	H	L	H	L	Write Y operand to Adder - Function:Subtract
L	H	H	L	H	H	Write Y operand to Adder - Function:Float to Fix
I	H	H	H	L	L	Write Y operand to Adder - Function:Fix to Float
L	H	H	H	L	H	Write Y operand to Adder - Function:Check Status
L	H	H	H	H	L	Write Y operand to Adder - Function:Divide
L	H	H	H	H	H	NO OP
H	L	H	L	L	L	Read Adder result and status
L	H	L	X	L	L	Write X operand to Multiplier
L	H	L	X	L	H	Write X operand to Multiplier
H	L	L	X	L	L	Read Multiplier result and status
X	X	L	X	H	X	NO OP
L	L	X	X	X	X	Illegal Condition

FLOATING POINT BUS STATUS LINES

$\overline{\text{OVF}}$ Low on this line indicates the result is beyond the range of numbers which can be represented. All bits of the result set to 1.

$\overline{\text{UFL}}$ Low on this line indicates the result is smaller than the smallest number which can be represented. All bits except for the ZERO BIT (BIT 62) are set to 0.

NEG High on this line indicates the result of the operation was a negative number. It is the same as the negative bit of the mantissa.

$\overline{\text{ZER}}$ Low on this line indicates the result of the operation was zero. All bits of the result are zero.

$\overline{\text{DNM}}$ Low on this line indicates the Multiplier is done and ready for new input.

$\overline{\text{DNA}}$ Low on this line indicates the Adder is done and ready for new input.

$\overline{\text{DBYZ}}$ Low on this line indicates a divide by zero error in the floating point Adder/Subtractor/Divider.

FLOATING POINT MULTIPLIER (FPM)

The Floating Point Multiplier is a very high speed microprogrammed logic board designed to exclusively perform all floating point multiplication within a node processor.

The FPM is connected to the CPU of a node via the Floating Point Bus Interface (FPBI). The interface buffers, formats and stores up to 256 72-bit floating point values for processing by the CPU or the Floating Point units. (See Floating Point Bus Interface.)

The FPM has three registers which are accessed via the FPBI. They are the X operand input register, the Y operand input register, and the Result and Status output register. The floating point control bus signals required to access these registers are:

$\overline{\text{RDFP}}$	$\overline{\text{WRFP}}$	ADD/$\overline{\text{MULT}}$	FPRI	FPRO	
H	H	X	X	X	No Op
X	X	H	X	X	No Op (FPA)
H	L	L	L	L	Load X operand of multiplier
H	L	L	L	H	Load Y operand of multiplier
L	H	L	L	L	Read Result and Status of Multiplier

The FPM is loaded under control of the node processor CPU. The order of operand loading is important. The X-operand is ordinarily loaded first. Upon the loading of the Y operand, the multiplier begins execution. On a succeeding multiply, if the X operand does not change, only the new Y operand need be loaded. The multiplier will proceed using the old X operand and the new Y operand. The result returned is a 72-bit product with appropriate status bits.

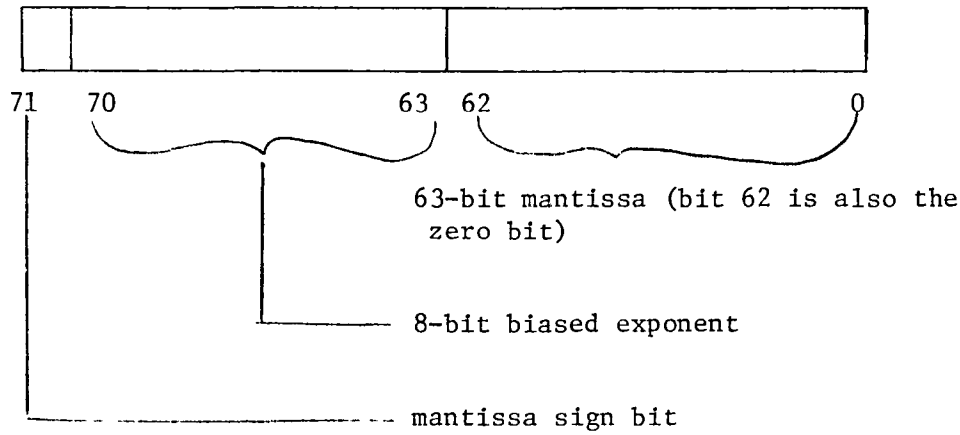
Assembly Language Instructions

The floating point multiplier is under control of the node processor CPU. The instruction FMUL is the only instruction which uses the FPM.

Multiplication Algorithm

The floating point numbers have the representation of 1 sign bit, an 8-bit exponent with a bias of 128 and a 63-bit mantissa for a total of 72 bits.

This format provides a range of 10^{-37} to 10^{38} with 19 digits of precision. In all cases the floating point inputs are normalized numbers. Also, if the exponent is zero (-128_{10}) then the number is zero. This eliminates gradual underflow or operation with vanishing numbers



The floating point multiplication is done in two relatively independent processes. One process determines the sign and exponent of the result, the other process determines the mantissa. The two processes interact when the final mantissa may need to be normalized, thereby changing the exponent of the result. The multiply algorithm is flowcharted in Figure 17.

The sign of the result is 1 (negative) if, and only if, the signs of the inputs are not equal. The exponent is found by adding the input exponents

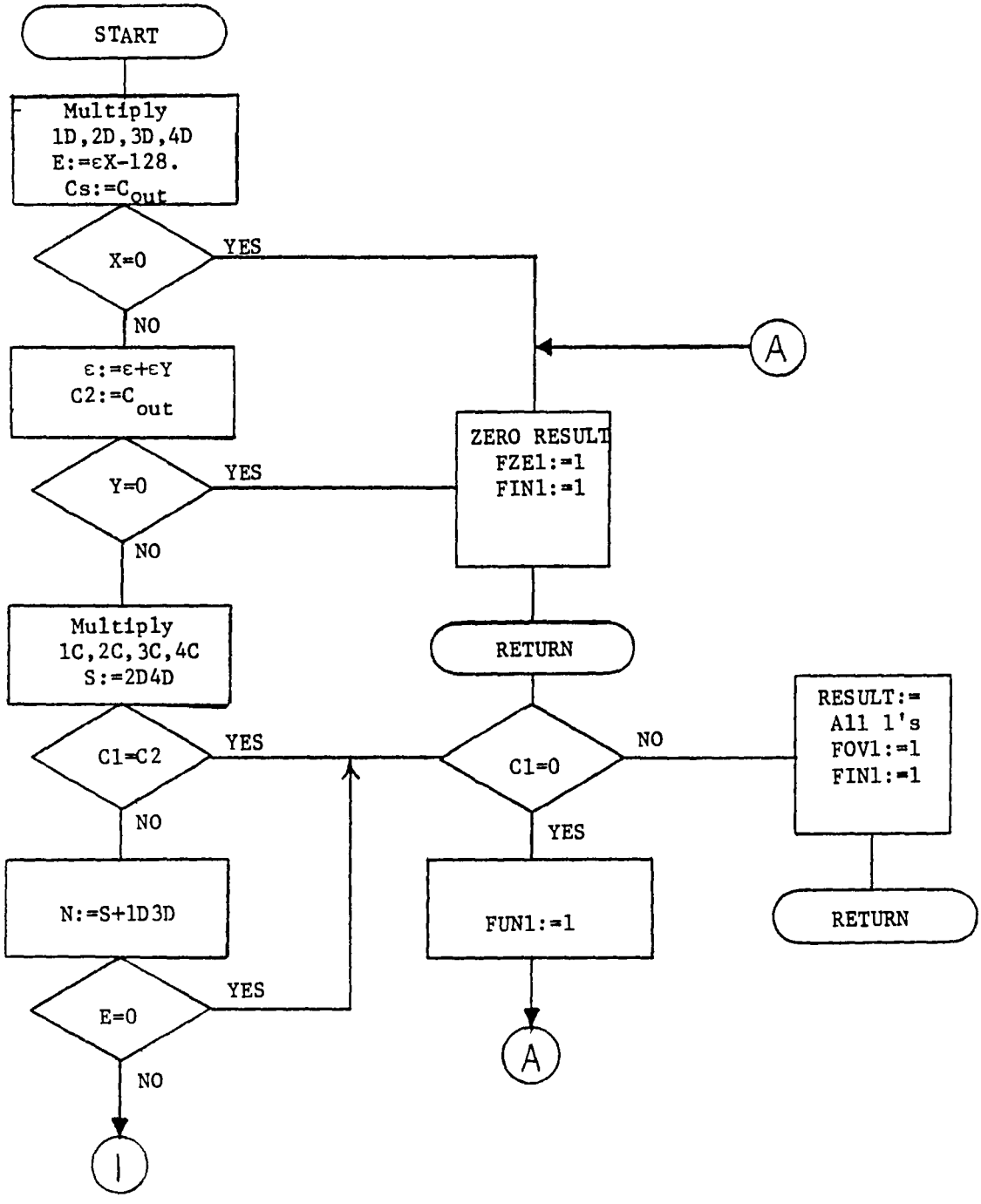


Figure 17 Floating Point Multiply Flowchart

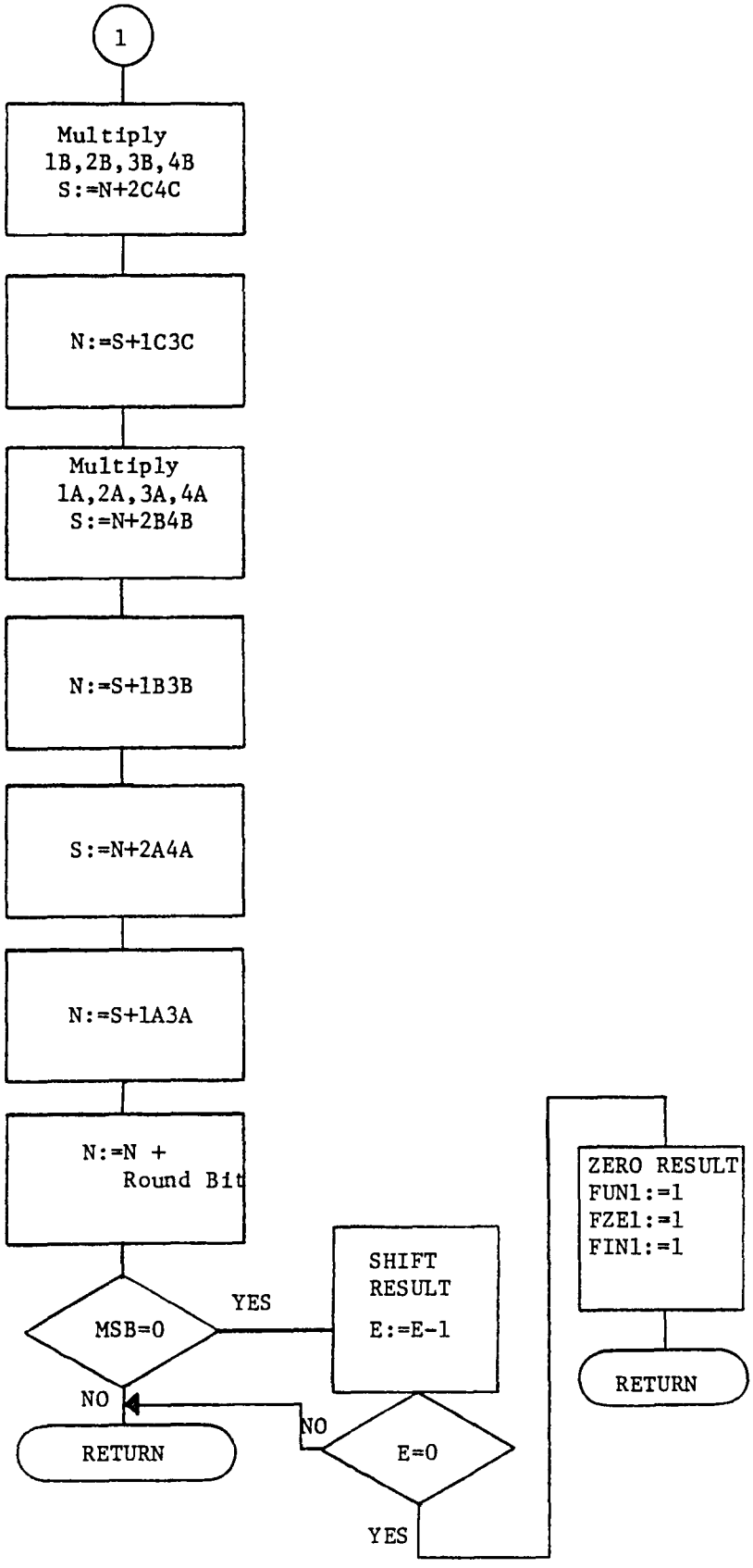
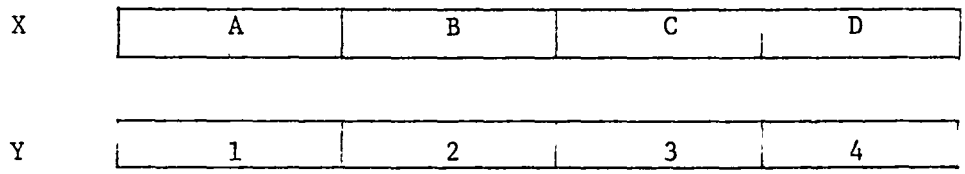


Figure 17 Continued

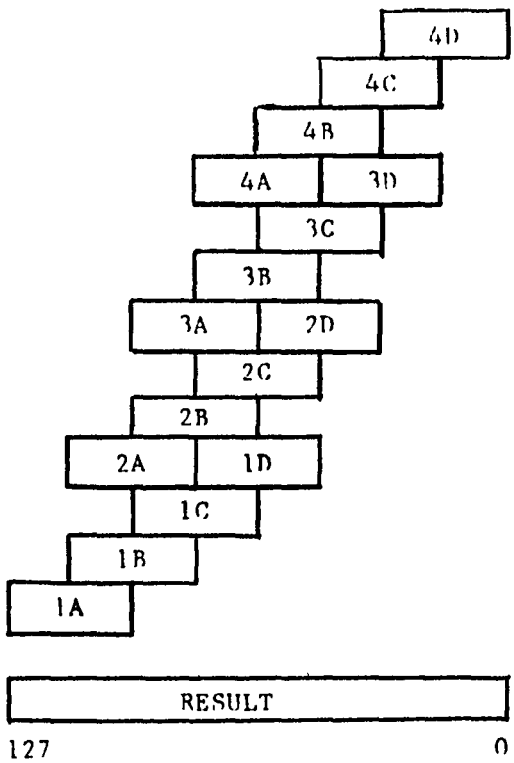
and subtracting the bias (128_{10}). After the result has been normalized underflow will occur if the exponent is below the minimum allowed. Overflow will occur if the exponent is greater than the maximum.

The calculation of the result mantissa is done by finding the sum of 16 partial products, each 32 bits, to form a 128-bit result. The result is then normalized, rounded and truncated to 63 bits.

The X and Y input mantissas are represented by four 16-bit fields. The 63-bit input mantissas are left justified within these 64 bits.



The 32-bit result of a 16 x 16 multiply is represented by the 2 16-bit fields multiplied. The 16 32-bit partial products are added to form a 128-bit result. The partial products must have their LSB aligned with the proper bit in the 128-bit result before adding. The alignment of the partial products may be visualized as:



The hardware does not maintain the full 128-bit result. The lower 64 bits are used to set guard and sticky bits are used when rounding is accomplished. (See round and normalize hardware.)

F.P. MULTIPLIER HARDWARE

The FPM hardware consists of: a microprogram controller and PROMS, two operand input registers, an exponent ALU, four array multipliers, a mantissa ALU, a mantissa shifter, round and normalization logic and an output buffer. See Figure 18.

Microprogram Controller and PROMs

The FPM has its own microprogram contained in PROM. The microprogram counter is a binary up counter with preset and reset capabilities. The counter is reset to zero when the Y operand is loaded. The counter normally sequences through the microcode. Certain microinstructions allow conditional or unconditional presetting of the counter which causes branching to different sections of the microprogram. At the end of the multiply microroutine, the counter is disabled until the new operand(s) is(are) loaded.

Operand Input Registers

The input registers are positive edge triggered registers which are addressed and loaded under the control of the CPU. There are 2 72-bit input registers. One holds the X operand; the other holds the Y operand. Also on the input of the FPM is some decoding logic which determines from the CPU FP control signals whether the X operand is to be loaded, the Y operand is to be loaded or the Result is to be read.

Exponent ALU

The exponent ALU consists of an eight bit ALU, an output register, an output buffer and two selectable constant inputs. The A input of the exponent ALU may be either the exponent of the X operand or the output of the latch or

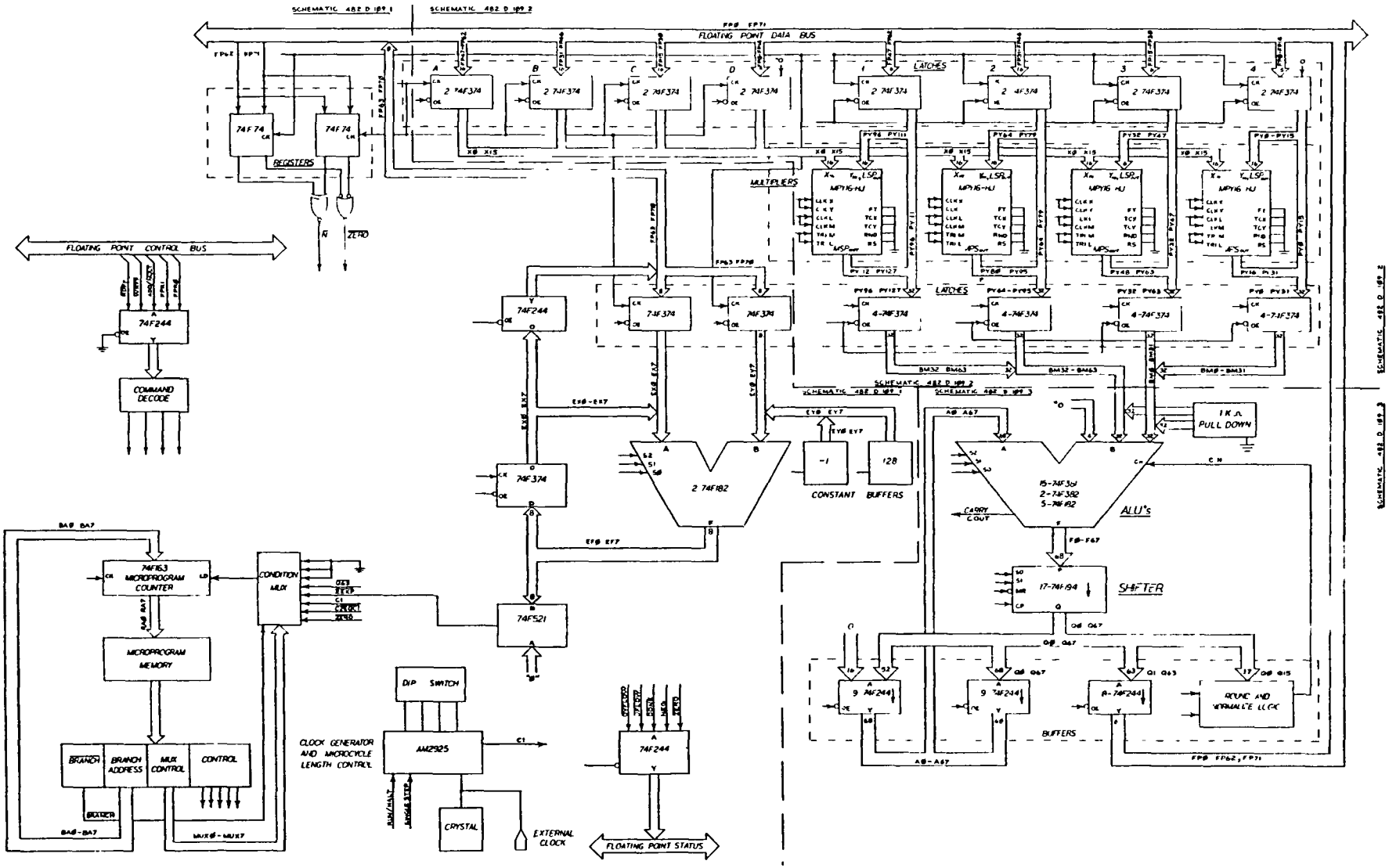


Figure 18 NASA/Lewis SDS Node Processor Floating Point Multiplier Block Diagram

the adder output. The B adder input may be the exponent of the Y operand, the constant -1, or the constant -128.

Array Multipliers

There are four 16 x 16 array multipliers which are used to generate four 32-bit partial products simultaneously. Four multiplies are done in each multiplier to compute all 16 partial products. The bits generated by the multipliers are latched in four separate 32-bit tristate registers.

Mantissa ALU

The mantissa ALU is a 68-bit wide ALU. The ALU output is loaded into the parallel load input of a 68-bit shifter. The A input of the adder may be either the output of the 68-bit shifter, or the output of the 68-bit shifter right shifted 16 bits.

The B input of the ALU selects between two pairs of latched multiplier outputs.

Mantissa Shifter

The mantissa shifter latches the output of the 68-bit adder. The shifter is used to left shift the result if normalization is necessary.

Round and Normalize Logic

Since there are more bits computed than are retained for a final result, it is necessary to round the result. All results are rounded to the nearest expressible value, with rounding to even if the value is exactly between the 2 possible representations. In this scheme, a guard bit and a sticky bit are necessary. The result of a mantissa multiplication is 64 bits. The LSB of the intermediate result is designated as the rounding bit. The guard bit is the bit immediately to the right of the LSB which would normally be lost. In the event the intermediate result must be left shifted to normalize, the guard bit is shifted into the LSB of the result and becomes the rounding bit.

The sticky bit and guard bit are used to correctly determine which direction to round when done. The sticky bit "remembers" if there were any bits set in the low order 63 bits of the 128-bit result. These low order bits are not carried through the computation. Instead, during each of the four multiply steps, the low order 16 bits are passed to the guard and sticky bit logic. This logic does the following:

1. Logical OR of guard bit with sticky bit
2. Logical OR 15 least significant bits with sticky bit
3. MSB of 16 bits becomes new guard bit

In the round step a 1 is added to the mantissa if and only if

$$1 = (\text{Bit } 63) \cdot (\text{Bit } 0) \cdot (\text{Guard} + \text{Sticky}) + (\overline{\text{Bit } 63}) \cdot (\text{Guard})(\text{Sticky})$$

FLOATING POINT ADDER/SUBTRACTOR/DIVIDER (FPA)

The Floating Point Adder is a high speed microprogrammed logic board designed to perform floating point addition, subtraction and division. The FPA also will convert a floating point number to fixed point, a fixed point number to floating point, and set the appropriate status for a single input.

The FPA is connected to the CPU of a node via the Floating Point Bus Interface (FPBI). The interface buffers format and store up to 256 72-bit floating point values for processing by the CPU or the Floating Point units (see Floating Point Bus Interface).

The FPA has three registers which are accessed via the FPBI. They are the X-operand input register, the Y-operand input/command register, and the result/status output register. Floating Point control bus signals to access these registers are:

$\overline{\text{WRFP}}$	$\overline{\text{RDFP}}$	ADD/$\overline{\text{MULT}}$	FPR2	FPR1	FRRØ	
H	H	X	X	X	X	No Op
L	H	L	X	X	X	No Op (FPM)
L	H	H	L	L	L	Write X-Operand to FPA
L	H	H	L	L	H	Write Y-Operand to FPA Function: ADD
L	H	H	L	H	L	Write Y-Operand to FPA Function: SUBTRACT
L	H	H	L	H	H	Write Y-Operand to FPA Function: FLOAT to FIX
L	H	H	H	L	L	Write Y-Operand to FPA Function: FIX to FLOAT
L	H	H	H	L	H	Write Y-Operand to FPA Function: CHECK STATUS
H	L	H	L	L	L	Read FPA Result and Status

FP Adder/Subtractor and Divider Hardware

The FPASD hardware consists of: a microprogram controller and proms, an exponent ALU, a mantissa ALU, a mantissa shifter, and output buffers and latches. See Figure 19.

Microprogram Controller and Proms

The FPASD has its own microprogram contained in PROM. The microprogram counter is a binary up counter with preset and reset capabilities. The counter is preset to a special address in the lower address space of the microcode PROM upon the loading of the Y-operand. The preset address is determined by the FP register control bits. The microprogram then executes the instructions for the proper algorithm. Certain microinstructions allow conditional or unconditional branching to different sections of the microprogram. Otherwise, microprogram execution is sequential.

At the end of a microprogram, the counter is disabled until the new operand(s) is (are) loaded.

Operand Input Registers

The input registers are positive edge triggered registers which are addressed and loaded under control of the CPU. There are two 72-bit input registers. One contains the X-operand while the other contains the Y-operand. Also on the input of the FPASD is some decoding logic which determines from the CPU FP control signals whether the X-operand is to be loaded, the Y-operand is to be loaded, or the Result is to be read.

Exponent ALU

The exponent ALU consists of an 8-bit ALU, four registers, and a 8-bit counter. The first register is on the output of the ALU. The output of the first register feeds the 8-bit counter, a second register which has its output driving the A input of the ALU, and a third register used to hold the exponent result and drive the FP bus. The fourth register is on the outputs of the 8-bit counter and drives the B inputs of the ALU.

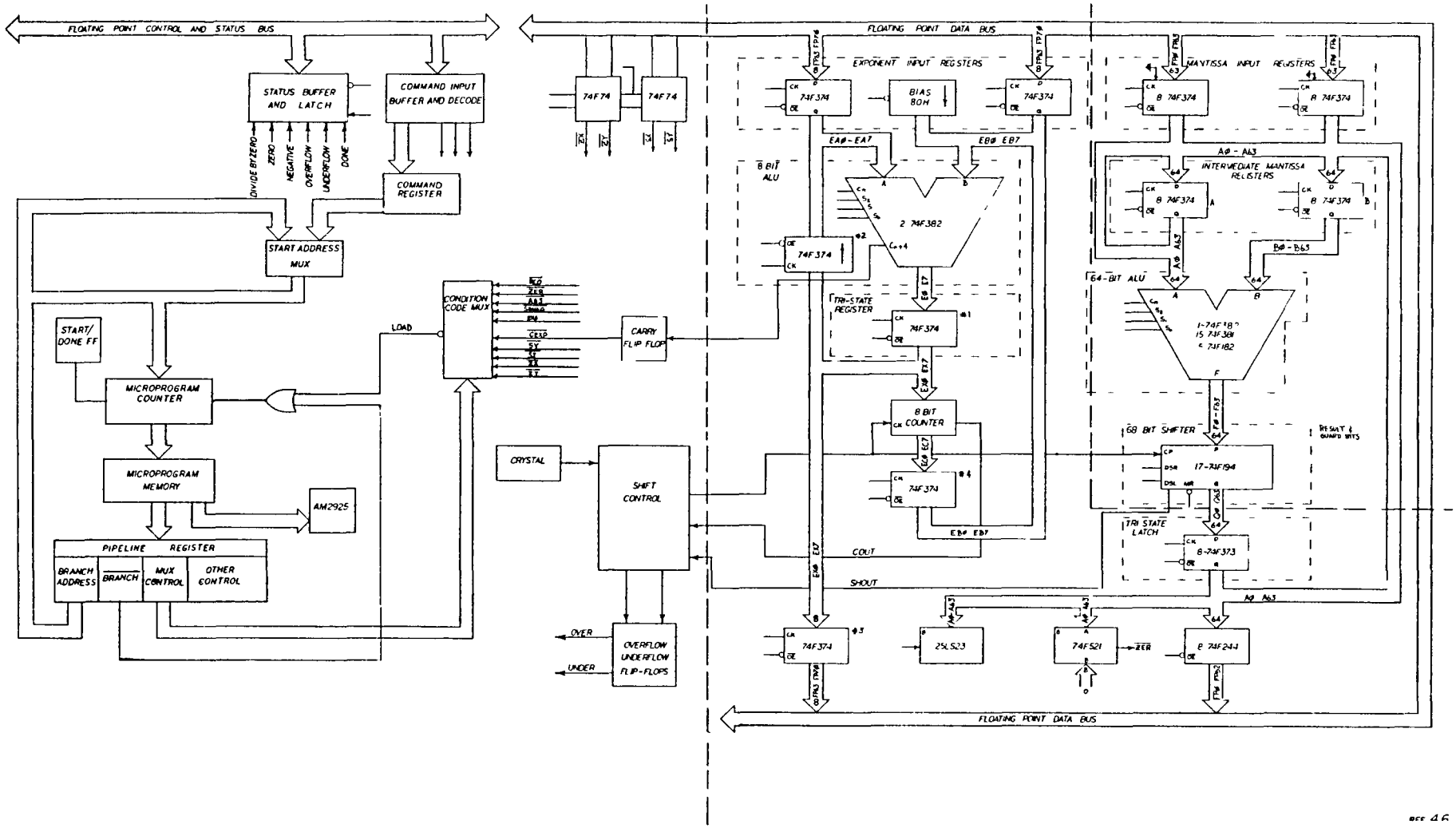


Figure 19 NASA/Lewis SDS Node Processor Block Diagram Floating Point Adder/Subtractor/Divider

In other words, input A to the ALU may come from either the X input register or register number 2. The B input of the ALU may come from either the Y input register or register number 4.

Mantissa ALU

The mantissa ALU consists of two 64-bit latches, a 64-bit ALU, a 68-bit shifter, a 64-bit transparent latch, a 64-bit zero detect circuit, and a 72-bit shifter.

Either of the two 64-bit intermediate mantissa registers may be loaded from the X operand input register, the Y operand input register, the transparent latch, the output of the A register itself, or the 72-bit shifter.

The ALU A input may come from either the X or Y operand input registers, the transparent latch, or the A intermediate mantissa register. The ALU B input may only come from the B intermediate mantissa register.

The input of the 68-bit shifter may only come from the 64-bit ALU. The input of the 64-bit latch may only come from the 68-bit shifter.

The 72-bit shifter, the zero detect circuit, and the 64-bit output buffer all have the same inputs as the ALU A input.

Assembly Language Instructions

FLOATING POINT ADD
FLOATING POINT SUBTRACT
FLOATING POINT DIVIDE
FIX TO FLOAT
FLOAT TO FIX
STATUS CHECK

Floating Point Addition Algorithm

Floating point addition is performed in the Floating Point Adder/Subtractor/Divider according to a two's complement addition algorithm described below. X and Y are the floating point input operands and Z is the sum. The flowchart for this algorithm is shown in Figure 20.

1. Compare X-operand and Y-exponent. If Y-exponent is greater than X-exponent swap X and Y so that the larger value is in X. If the exponents are equal, leave X and Y alone.
2. Subtract the exponents so that D = X-exponent - Y-exponent. If D is greater than or equal to 63 the answer is X and the procedure may be stopped, otherwise continue.
3. Convert the signed magnitude mantissa to two's complement form.
4. Shift Y-mantissa to the right D times.
5. Perform a two's complement addition of the two mantissas. Shift the result one place to the right and increment the exponent if a carry is generated.
6. Convert the two's complement result to sign magnitude form.
7. Normalize the result by shifting the mantissa left until a 1 appears in the most significant bit. Decrement the exponent for each shift.
8. Round and latch result.

Floating Point Subtraction Algorithm

Floating point subtraction differs only slightly from floating point addition. If Z = X-Y, change the sign of Y and proceed with steps 1 through 8 of the floating point addition algorithm. The flowchart for the algorithm is shown in Figure 21.

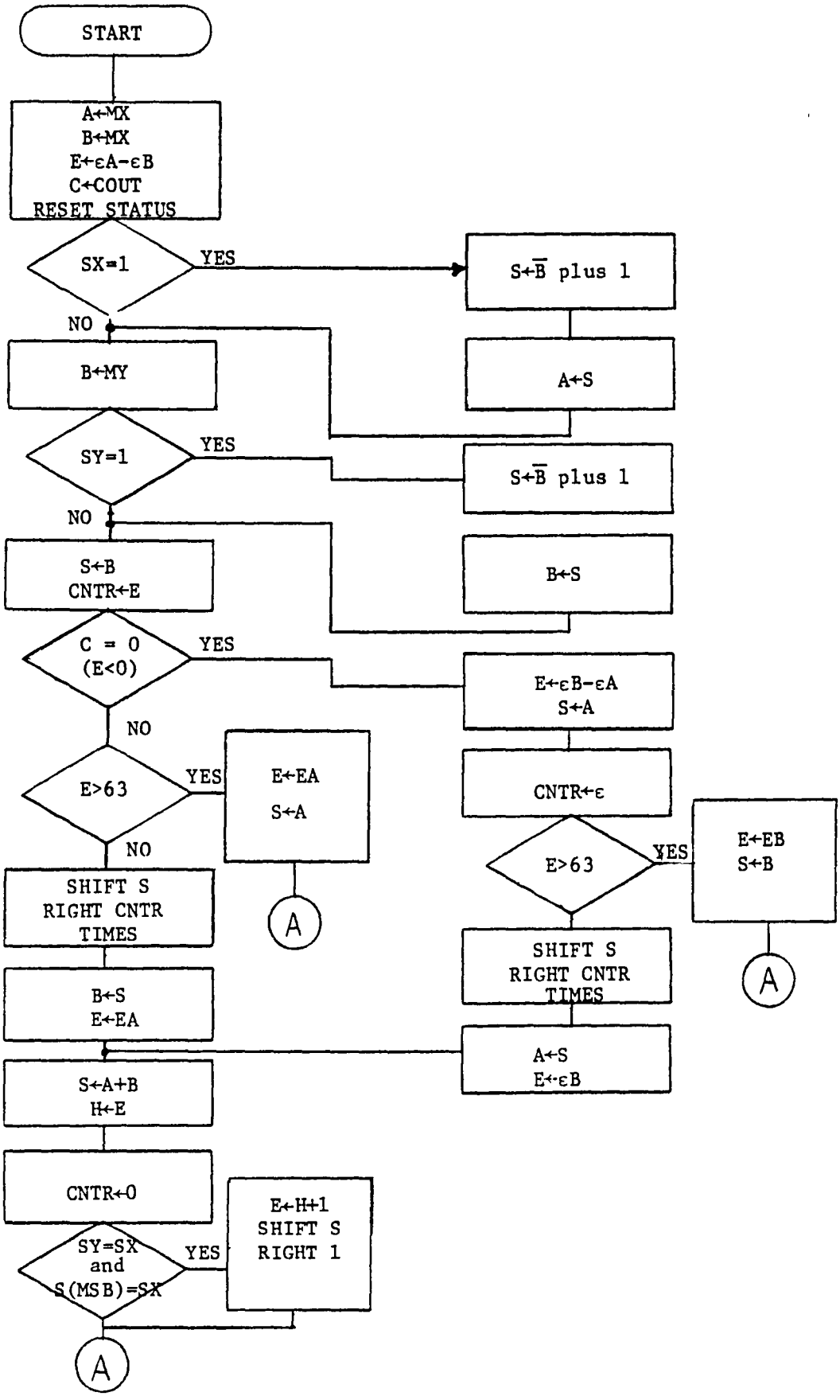


Figure 20 Floating Point Addition Flowchart

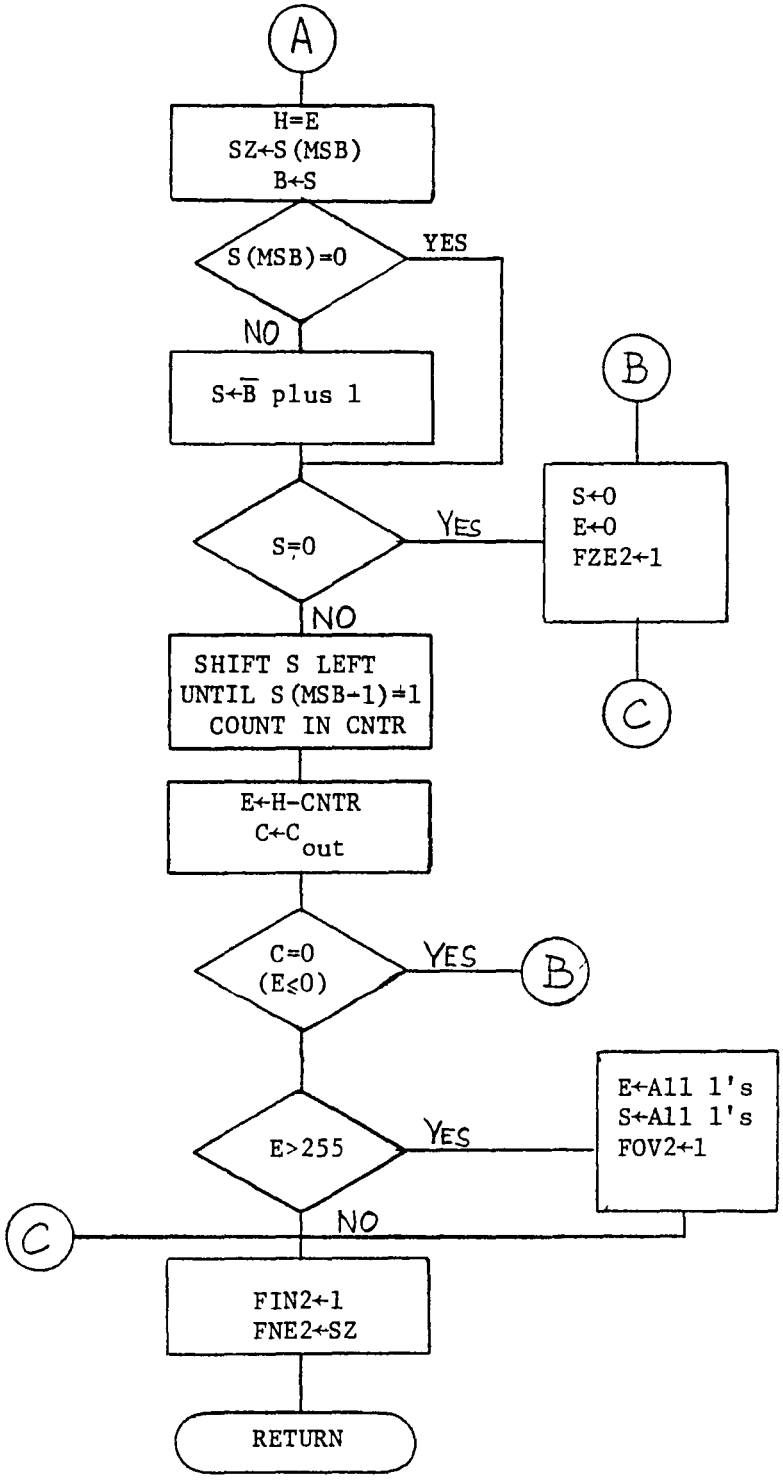


Figure 20 Continued

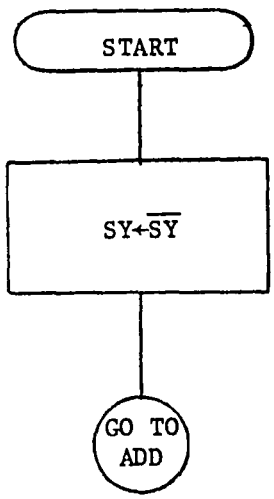


Figure 21 Floating Point Subtraction Flowchart

Floating Point Division Algorithm

Floating point division is performed in the Floating Point Adder/Subtractor/Divider according to a non-restoring binary division algorithm described below. The divisor is X, the dividend Y and the quotient is Z. The flowchart for the algorithm is shown in Figure 22.

1. Subtract the exponents so that Z exponent = X exponent - Y exponent + bias.
2. Compare the mantissas. If X mantissa is greater than the Y mantissa shift X mantissa to the right one bit and increment Z-exponent.
3. Set counter I to 62. Clear Z-mantissa (Z mantissa = 0).
4. If X-mantissa equals zero, then stop.
5. Perform subtraction X = 2X-Y.
6. Test result of subtraction. If negative go to step 10, else continue.
7. Set bit I in quotient to 1 (Z(I) = 1).
8. Decrement I.
9. If I does not equal zero, go to step 4, else stop.
10. Add X = 2X+Y.
11. Decrement I.
12. If I does not equal zero, go to step 6, else stop.

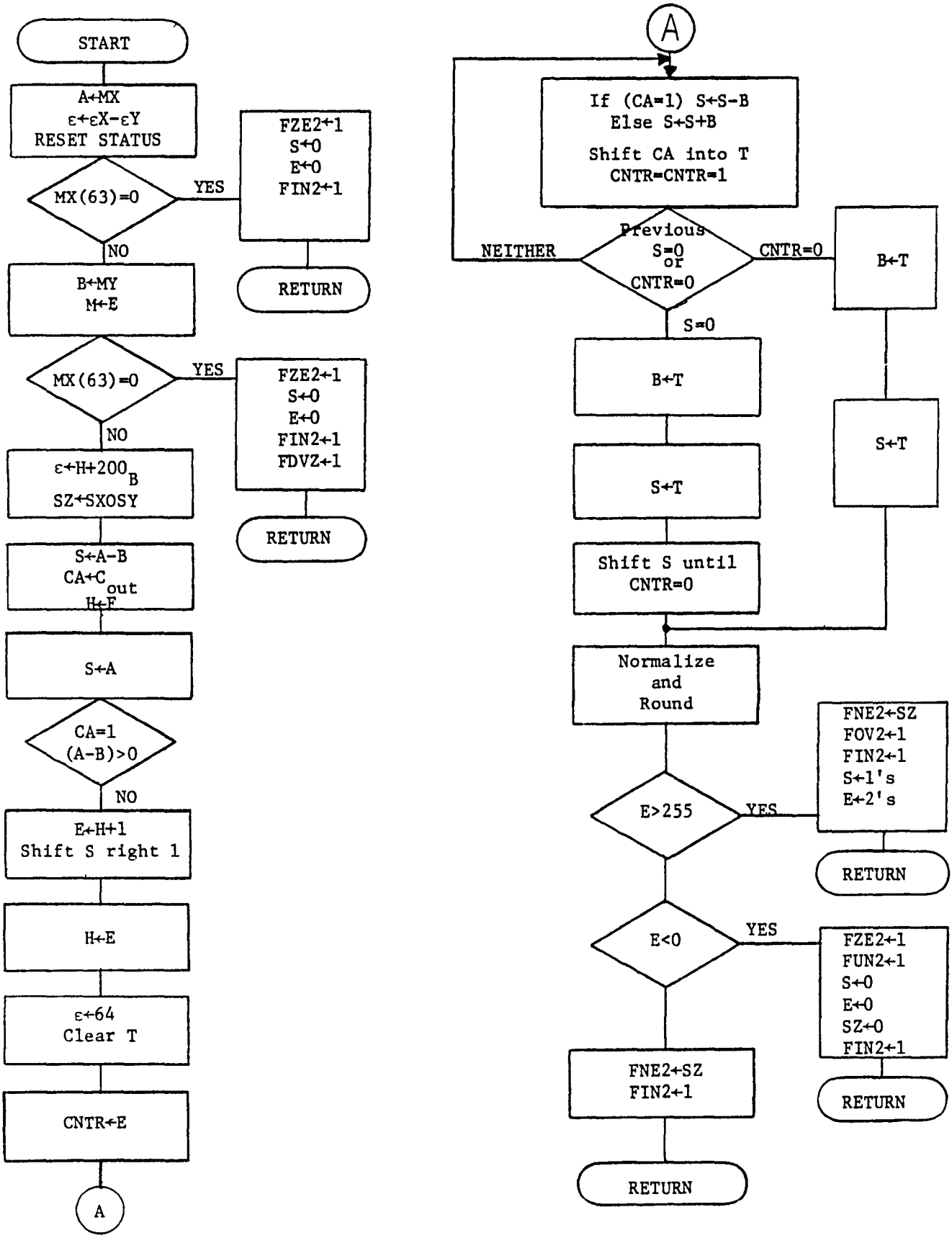


Figure 22 Floating Point Division Flowchart

HARDWARE ASSESSMENT

The system architecture of an array of node processors controlled by a central minicomputer was chosen. The node processor design was based upon the computational requirements of a structural dynamics simulator. A microprogrammable bit-slice architecture allowed an extremely powerful custom instruction set. Additional hardware was designed to greatly speed up floating point calculations.

Every node processor may access a large dynamic memory. Nearest neighbor communications have been implemented for efficient interprocessor communications.

The dynamic memory, CPU to floating point unit interface, and two floating point units, were designed up through schematic diagrams. The CPU and the communications interfaces were designed through block diagrams.

The hardware architecture chosen is as powerful as is possible while economically feasible with today's technology. As upgraded versions of the circuitry become available, the speed to cost ratio will increase. Waiting for future advancement poses few advantages. New products are rarely ready on schedule. The hardware suggested here is available now to provide an extremely powerful and useful tool for structural dynamics simulation.

DISCUSSION OF RESULTS

The approach taken in this design of a Digital System for Structural Dynamics Simulation is innovative. From a hardware standpoint, the system takes advantage of decreasing costs and increased computational power of state-of-the-art digital technology. The software associated with this system is of necessity state-of-the-art. The main concepts of value in the simulation application are the segmentation of the problem (into 125 equal parts), a custom instruction set tailored for simulation, and the high speed of the computing hardware.

At this point it is suggested that the detailed design of a node processor be completed and a prototype constructed. The instruction set should be microcoded and small programs should be written to exercise the hardware/firmware. At the conclusion of this phase, the decision can be made as to purchase of a control minicomputer and subsequent production of the entire array of processors.

The technical risk of producing a single functioning node processor is not great and results primarily from the tedium of microcoding such a machine. Constructing the entire system represents a challenging problem in packaging, cooling, interconnecting, and testing.

SUMMARY OF RESULTS AND RECOMMENDATIONS

The subject of this report has been the design of a Digital System for Structural Dynamics Simulation.

The results of this program can be summarized as follows:

1. A search of the field of parallel processing/simulation was done to discover work which would be a duplication of effort of this program. No such duplication was found.
2. The principles of simulation modeling methods were explored and a method (Runge-Kutta) chosen for this class of problems.
3. The architecture of an array of processors was conceived as the best possible solution to the simulation problem.
4. The node processor architecture of bit-slice microprogrammed CPU, large dynamic memory and custom floating point hardware was chosen.
5. The floating point hardware and dynamic memory were designed to the detailed schematic stage. The CPU was designed to the block diagram level.
6. The instruction set was designed and flowcharted. This information is contained in the Node Processor Instruction Set Reference and the Microcode Flowcharts for the Node Processor Instruction Set provided to NASA as a separate report.
7. System software requirements were outlined.

For meaningful continuation of this program, additional effort in the areas below is recommended.

1. A sample structural dynamics problem should be developed, its solution coded and run on a main frame computer.
2. The hardware and firmware (microcode) for a single node processor should be prototyped.
3. Segments of the sample problem should be run on the node processor with results being checked back to the main frame solution.

4. The control minicomputer should be specified and software design started for the system.

5. The debugged prototyped node processor can then be produced in quantity. Packaging and interfacing to the central minicomputer followed by system checkout will complete the program.

REFERENCES

1. Burden, R., Faines, J. and Reynolds, A., "Numerical Analysis", Prindle, Weber & Schmidt, Boston, Mass., 5th Printing, August 1980.

2. Chatsworth, A. E., "An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family", Computer, Vol. 14, No. 9, Sept. 81, pp. 18-27.

3. Advanced Micro Devices, "The Am 2900 Family Data Book", Advanced Micro Devices, California (1979).

APPENDIX I

Summary of Current Similar Simulation Programs

SUMMARY OF CURRENT SIMILAR SIMULATION PROGRAMS

	ORGANIZATION	PURPOSE OF WORK	APPROACH	PRESENT STATUS
1.	Univ. of Wisconsin	Solution of partial differential equations	Net of computers. Nearest neighbor (2 dimensional array). Global bus structure	Study Stage.
2.	NASA Ames	Calculation of three-dimensional flows for aircraft.	Use of parallel processing concepts to get 10^9 FLOPS. Architecture not based on physical problem.	System design.
3.	NASA Langley	Solution of static finite element equation.	Array of asynchronous microprocessors. Each connected to 12 nearest neighbors for communication of displacements. Iterative solution.	Four node system operating. Thirty-six node prototype under development.
4.	Goodyear Aerospace	Processing of satellite imagery.	Use of 128 x 128 array of processors to process simultaneously bit serial data. Each connected to four nearest neighbors. Custom VSLI CMOS/SOS chips are used for non-memory portions of each processor.	To be completed in 1982.
5.	Rensselaer Polytechnic Institute	Assessment of chip technology as related to structural engineering.	Evaluation of current and future chip usage in numerical algorithm evaluation. Prediction of impact of chip technology on numerical analyses.	Started in 1980. To be completed in 1982.

APPENDIX II

Summary of Relevant Papers Describing Parallel Processing Simulation Procedures

SUMMARY OF RECENT PAPERS DESCRIBING PARALLEL PROCESSING

	REFERENCE	COMPANY/ ORGANIZATION	KEY WORDS	SUBJECT
1.	Baskett, F. and A. J. Smith, "Interference in Multiprocessor Computer Systems with Interleaved Memory," ACM, Volume 19, No. 6, June 1976, pp. 327-334.	Stanford University, University of California, Berkeley	MEMORY INTERFERENCE, MULTIPROCESSING, INTERLEAVED MEMORY TRACE DRIVEN SIMULATION.	Analyzes the memory interference caused by several processors simultaneously using several memory modules. Results are computed for a simple model.
2.	Baudet, G. M., "Asynchronous Interative Methods for Multi-Processors" Journal ACM, Volume 25, No.2, April 1978, pp 226-244.	Carnegie-Mellon University, Pittsburgh	ASYNCHRONOUS ALGORITHMS, ASYNCHRONOUS MULTIPROCESSORS, PARALLEL ALGORITHMS, INTERATIVE METHODS, CHAOTIC RELAXATION, ANALYSIS OF ALGORITHMS	Asynchronous iterative methods presented for solving a system of equations. Conditions given to guarantee convergence. Advantages of purely asynchronous methods.
3.	Bhandarkar, D. P., "Some Performance Issues in Multiprocessor System Design," IEEE Trans. Computers, Volume C-26 No. 5, May 1977, pp 506-11.	Australian National University, Canberra, Australia	MEMORY INTERFERENCE, MEMORY INTERLEAVING, MULTIPROCESSORS.	Guidelines for multiprocessor system architect. Preferred design alternatives and/or tradeoffs.
4.	Enslow, P. H., "Multiprocessor Organization—A Survey," Computing Surveys, Volume 9, No. 1, Mar. 1977, pp 103-129.	Georgia Institute of Technology, Atlanta	COMPUTER SYSTEM ORGANIZATION, CONCURRENT OPERATIONS, INTERCONNECTION SUBSYSTEMS, MULTIPROCESSOR OPERATING SYSTEMS.	Time-shared buses, crossbar switch matrix, multibus/multiport memories, interconnection systems discussed. Three operating systems master-slave, separate executive for each processor, symmetric treatment of all processors reviewed.
5.	Kafura, D. G. and V. Y. Shen., "Tasks Scheduling on a Multiprocessor System with Independent Memories," SIAMJ Computing, Vol. 6, No. 1, Mar. 1977, pp 167-187	Iowa State University, Purdue University	SCHEDULING, ALGORITHMS, DETERMINISTIC MODELS, WORST-CASE BOUNDS, MEMORY CONSTRAINTS.	Scheduling strategies evaluated for system of identical processor with a private memory.

SUMMARY OF RECENT PAPERS DESCRIBING PARALLEL PROCESSING
(Continued)

	REFERENCE	COMPANY/ ORGANIZATION	KEY WORDS	SUBJECT
6.	Kinney, L. L., and R. G. Arnold, "Analysis of a Multi-Processor System with a Shared Bus," Conference Proceedings 5th Ann. Symp. Computer Architecturs, Palo Alto, CA April 1978, pp 89-95.	University of Minnesota, Honeywell Corporation, Minneapolis	FIFO, QUEUE, MULTIBUS SYSTEM, FIFO SHARED-BUS.	Analysis of a multiprocessor system with shared-bus. Determining the processing power as the number of processors is increased.
7.	Kuznia, C. H., R. Kober, and H. Kopp, "SMS - A Structured Multimicroprocessor System with Deadlock-Free Operation Scheme" Conf. Proc. 3rd Ann. Symp. Computer Architecture, Clearwater, Florida, Jan. 1976, p. 122	SIEMENS AG Hofmannstr, Germany	DISTRIBUTED MEMORY, MULTIPROCESSOR, COMMUNICATIONS MEMORY	Multiprocessor system design for large systems of differential equations.
8.	O'Grady, E. P., "A Multiprocessor for Continuous System Simulation", Proceedings 1979 Int. Conf. Parallel Processing, Bellaire, MI, Aug. 1979, p. 306.	Arizona State University Temple	SIMULATION, INTER-PROCESSOR COMMUNICATION, BIT-SLICE MICROPROCESSOR BUS CONTROL PROCESSOR.	Simulation-oriented multiprocessor system employing a new concept in interprocessor communication is described. Parallelism in conjunction with address-mapping memories realize an efficient high-speed transfer mechanism.
9.	Pearce, R. C. and J. C. Majithia, "Upper Bounds on the Performance of Some Processor-Memory Interconnections," Proc. 1967 Int. Conf. Parallel Processing, Walden Woods, MI, Aug. 1976, p. 303.	University of Waterloo, Ontario	CROSS-POINT, TIME-SHARED BUS, PIPELINED LOOP, BINARY SWITCH.	Multiprocessor performance evaluated for cross-point, time-shared pipelined loop, and binary switch methods.

SUMMARY OF RECENT PAPERS DESCRIBING PARALLEL PROCESSING
(Continued)

REFERENCE	COMPANY/ ORGANIZATION	KEY WORDS	SUBJECT
10. Pearce, R. C. and J. C. Majithia "Performance Results for an M.I.M.D. Computer Organization Using Pipelined Binary Switches and Cache Memories," Proc. Inst. Electronic Engineers (England), Vol. 125, No. 11, Nov. 1978, pp. 1203-1207.	University of Waterloo, Ontario	PIPELINED PROCESSING, CACHE MEMORIES, M.I.M.D. ARCHITECTURE	Throughput performance with respect to variations in cache memory parameters, number of processors, processing time of a system in which a binary switch is used as the interconnection network.
11. Sastry, K. V. and R. Y. Kain, "On the Performance of Certain Multiprocessor Computer Organization," IEEE Trans. Computers, Vol. C-24, No. 11, Nov. 1975, pp. 1066-1074.	Sperry Univac Roseville, Minn., University of Minnesota	ANALYTIC MODELS, INSTRUCTION EXECUTION RATES, MEMORY CONFLICTS, MULTIPROCESSORS.	Performance of a multiprocessor system with different storage allocations for instructions and data with interleaving in the instruction space is presented.
12. Yang, Chao-Chih, "Gast Algorithms for Bounding the Performance of Multiprocessor Systems," Proc. 1976 Intl. Conf. Parallel Processing, Walden Woods, Mich., Aug. 1976, pp. 73-82.	University of Alabama Birmingham	PRECEDENCE PARTITION, PARTIALLY ORDERED TASKS	Proposing two types of scheduling for more efficient execution of a multiprocessor system.
13. Patel, Janak H., "Processor-Memory Interconnections for Multiprocessors," Conf. Proc. 6th Ann. Symp. Computer Architecture, Philadelphia, PA, April 1979, pp. 168-177.	School of Electrical Engineering, West Lafayette, IN	INTERCONNECTION NETWORKS, CROSSBAR SYSTEMS, DELTA NETWORKS	Interconnection networks proposed for processor to memory communication and multiprocessing system allows a direct link between any processor to any memory module.

End of Document